Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Институт радиоэлектроники и информационных технологий — РТФ

Ю. П. ПАРФЁНОВ

СРЕДСТВА УПРАВЛЕНИЯ И ЗАЩИТЫ ИНФОРМАЦИОННЫХ РЕСУРСОВ АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Учебное пособие

Министерство науки и высшего образования
Российской Федерации

Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

Ю. П. Парфёнов

Средства управления и защиты информационных ресурсов автоматизированных систем

Учебное пособие

Рекомендовано методическим советом
Уральского федерального университета
для студентов вуза, обучающихся
по направлению подготовки
09.03.01 — Информатика и вычислительная техника

Екатеринбург
Издательство Уральского университета
2020

УДК 004.056.5(075.8)
ББК 32.971.35-5я73
П18

Рецензенты: кафедра менеджмента и управления качеством Уральского государственного лесотехнического университета (завкафедрой д-р техн. наук, проф. *В. П. Часовских*);
директор ООО «ТЭКСИ-СОФТ» *Д. А. Виницкий*

Научный редактор — канд. техн. наук, доц. *К. А. Аксёнов*

Парфёнов, Ю. П.

П18 Средства управления и защиты информационных ресурсов автоматизированных систем : учебное пособие / Ю. П. Парфёнов ; Мин-во науки и высш. образования РФ. — Екатеринбург : Изд-во Урал. ун-та, 2020. — 120 с.

ISBN 978-5-7996-3088-1

Материалы пособия направлены на развитие компетенций, полученных студентами при изучении дисциплины базы данных в части использования средств управления данными при разработке ERP- и MES-систем. В пособии подробно рассматриваются средства для создания программных модулей серверной компоненты корпоративной информационной системы с использованием как структурного языка запросов, так и алгоритмических языков в Common Language Runtime. Особое внимание уделяется средствам разработки эффективных многопользовательских транзакционных систем. Изложение каждой темы сопровождается примерами использования и контрольными вопросами. На примере MS SQL SERVER излагаются модели и средства защиты данных и программ серверной компоненты от несанкционированного использования.

Библиогр.: 16 назв. Табл. 4. Рис. 19.

УДК 004.056.5(075.8)
ББК 32.971.35-5я73

ISBN 978-5-7996-3088-1

Основные сокращения

БД — база данных

КИС — корпоративная информационная система

ОС — операционная система

СУБД — система управления базами данных

API — application programming interface

BLOB — Binary Large Object (большой двоичный объект)

CLR — Common Language Runtime (общеязыковая среда выполнения программ)

DEK — Database Encryption Key (симметричный ключ для шифрования пользовательской базы)

DMK — Database master key (главный ключ базы Master)

ERP (Enterprise Resource Planning) — планирование ресурсов предприятия

MRP — Material Requirement Planning (планирование потребностей в материалах)

SMK — (Service Master Key) асимметричный ключ службы MS SQL SERVER

TDE — Transparent Data Encryption (прозрачное шифрование баз данных)

Введение

В пособии рассматриваются средства создания и защиты информационных ресурсов в автоматизированных системах предприятий.

Ресурсы предприятия объединяют структурированные данные, сосредоточенные в информационных базах, файлы различного формата и назначения, а также специальные программы обработки этих данных. Значительное внимание в пособии уделено средствам создания транзакционных систем, интегрирующих в общей базе двоичные объекты большого объема. Излагаются средства загрузки, хранения, обработки и защиты больших объектов при совместном доступе в многопользовательском режиме.

Применительно к MS SQL SERVER рассматривается создание хранимых в общей базе программных модулей на Transact-SQL и в управляемом коде.

Дается подробная информация об организации защиты данных базы от несанкционированного доступа и средствам шифрования данных. Пособие ориентировано на студентов бакалавриата, знакомых со средствами создания и использования баз данных.

1. Управление информационными ресурсами в транзакционных системах

Транзакционной принято считать информационную систему, которая обеспечивает гарантированное выполнение отдельных независимых заданий — транзакций. Если по причине отказа оборудования или программного обеспечения выполнить задание не удается, то оно просто не выполняется. В корпоративных ERP- и MRP-системах информационные ресурсы многих пользователей объединяются в общей базе данных. Для эффективного функционирования системы важно обеспечить независимую параллельную обработку информации в различных бизнес- и технологических процессах. Для независимого доступа и обработки информации в серверах баз данных предусмотрены средства создания и выполнения параллельных транзакций. В транзакциях обеспечивается необходимая защита обрабатываемых данных от действия других транзакций путем установки различных блокировок или созданием версий (копий) обрабатываемых данных.

1.1. Транзакции реляционной базы данных

Транзакция — набор команд SQL, выполняющий запросы или изменения данных, приводящие базу в новое допустимое состояние. Транзакция должна либо выполниться целиком,

либо целиком не выполниться, сохраняя данные в исходном состоянии. Данные, обрабатываемые в транзакции, являются объектом защиты как от отказов сервера, так от действий других параллельно работающих транзакций (пользователей) [1].

Таким образом, с помощью транзакций решаются две проблемы обработки БД:

- сохранение целостности данных базы в условиях возможного отказа программы или оборудования;
- защита данных при их одновременной обработке несколькими пользователями.
- Стандарт SQL/92 установил требования к транзакциям, называемые ACID: Atomicity, Consistency, Isolation, Durability.
- **Атомарность** (Atomicity), или неделимость транзакции — изменения, выполненные транзакцией, целиком принимаются (фиксируются в базе) или целиком отменяются (откат базы к исходному состоянию). Дополнительными установками в параметре подключения SET XACT_ABORT OFF можно задать возможность частичного исполнения транзакции.
- **Согласованность** (Consistency) — данные в БД до начала транзакции и после ее выполнения должны соответствовать всем ограничениям базы, в том числе и ограничениям ссылочной целостности. Однако в процессе выполнения транзакции ограничения могут быть нарушены.
- **Изолированность** (Isolation). Изменения в базе, вносимые одновременно выполняемыми транзакциями, должны быть независимы друг от друга. Результат каждой изолированной транзакции должен определяться состоянием БД до ее начала и выполняемым действием.
- **Устойчивость** (Durability). Результаты выполненной до конца транзакции должны гарантированно сохраняться в базе на диске.

Далее в примерах использования транзакций рассматривается учебная база данных печатных изданий (PUBS). В базе содержатся данные об авторах, написанных ими книгах и данные о продажах книг. Схема БД представлена на рис. 1.1.

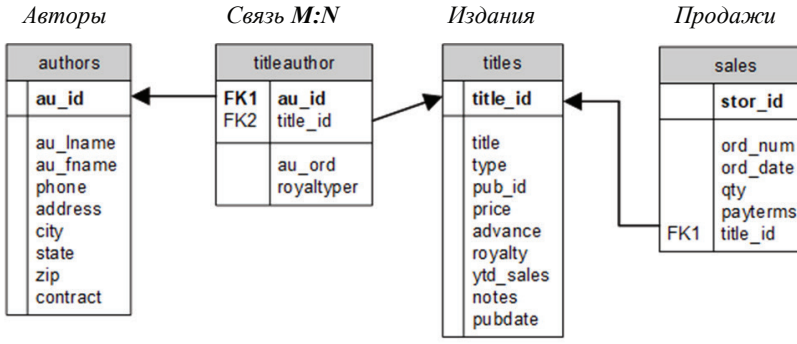


Рис. 1.1. ER-диаграмма учебной базы PUBS

Рассмотрим примеры применения транзакций.

Пусть в базе изданий необходимо изменить значения ключа (au_id) в записи одного из авторов. Такое изменение требует выполнения двух операторов UPDATE (для таблицы authors и таблицы titleauthor). Однако если при выполнении второго оператора произойдет прерывание программы по ошибке или «зависание» сервера, вторая таблица не будет обновлена и соответствие между строками таблиц нарушится. Значит, наборы операторов, выполняющих одну задачу, должны выполняться в одной транзакции. Программа замены значения ключа автора также должна предусматривать наличие ограничения ссылочной целостности типа «Restrict» между строками таблиц authors и titleauthor, которое не позволит изменить ключ в поле au_id ни в одной из таблиц. Поэтому перед выполнением первого обновления (UPDATE) необходимо отключить это ограничение командой ALTER TABLE … NOCHECK … . Полная

транзакция обновления ключей ссылочной целостности должна состоять из следующих действий.

1. Отключение ограничения ссылочной целостности (ALTER TABLE …NOCHECK …)
2. Выполнения первого оператора UPDATE, который, изменив ключи в одной таблице, приведет данные к несогласованному состоянию (согласованность должна быть обеспечена после выполнения транзакции).
3. Обновление (UPDATE) ключа в другой таблице, которое вернет соответствие данных.
4. Восстановить проверку ограничения по внешнему ключу au_id таблицы titleauthor оператором ALTER TABLE с опцией CHECK.

Другой пример — использование транзакций в запросах к БД.

Пусть из таблицы titles необходимо выбрать издания, имеющие цену выше, чем средняя цена издания. Для решения такой задачи надо сначала выполнить запрос, вычисляющий среднюю цену издания, а в следующем запросе воспользоваться найденным значением для отбора изданий. Если между этими двумя запросами другой пользователь изменит цену хотя бы одного издания, добавит или удалит издание, результат запроса будет неверным. Поэтому для защиты данных эти запросы надо выполнить в одной транзакции.

Реализация требований ACID для транзакций БД обеспечивается с помощью двух средств MS SQL SERVER:

– журнала транзакций,
– менеджера блокировок.

Журнал транзакций (transaction log) создается в отдельных файлах ОС. Запись изменений данных в журнал осуществляется до сохранения ее в файлах БД на диске. Поэтому журнал может быть использован для восстановления данных при отказе сервера. В журнале для каждой транзакции сохраняются:

– момент времени выполнения (начало и окончание) транзакции. Время используется для повторения цепочки действий в базе в хронологической последовательности;
– старые (до изменения) и новые (измененные транзакцией) записи или страницы базы данных;
– признак завершения транзакции;
– отметка о фиксации результата транзакции на диске в файлах БД.

Информация журнала используется при перезапуске сервера в случае его отказа следующим образом:

– производится изменение данных, выполненное в завершенных транзакциях, которое еще не было записано в БД на диске;
– выполняется откат — возвращение в исходное состояние измененных страниц в БД для незаконченных (незафиксированных) транзакций.

Кроме того, журнал может быть использован для внесения последних изменений базы при ее восстановлении с ранее созданной копии.

Обработку базы данных после повторного запуска, отказавшего («зависшего») сервера, демонстрирует рис. 1.2.

Рис. 1.2. Диаграмма параллельного выполнения транзакций

В момент отказа сервера результат транзакции T1 был записан на диск, поэтому данная транзакция не требует обработки.

Транзакции Т2 и Т4 были выполнены, но в журнале нет отметки о фиксации на диске, поэтому сервер в хронологическом порядке переписывает обновленные страницы из журнала в файлы БД на диске.

Транзакции Т3 и Т5 не была завершены, поэтому для них будет выполнен откат путем копирования из журнала старого состояния измененных страниц, а сами транзакции удалены из журнала. При восстановлении БД по журналу нет способа установить, какие именно транзакции не выполнены. Это можно сделать только по состоянию БД и результату работы приложений в момент отказа.

Различные серверы БД поддерживают разные типы транзакций. В MS SQL Sever реализованы три типа транзакций:

– автоматические (*фоновые*),
– неявные (*фоновые*),
– явные.

Автоматические транзакции создаются сервером БД по умолчанию. В режиме автоматических транзакций никаких действий по созданию транзакций пользователь не предпринимает. Каждая SQL-команда выполняется как отдельная транзакция. Если команда не была прервана, ее результат будет зафиксирован в базе, если команда вызвала исключение, сервер выполнит откат базы к состоянию, предшествующему началу команды. На время выполнения команды доступ к используемым записям автоматически блокируется сервером. В новом соединении транзакции выполняются в автоматическом режиме. Затем тип фоновых транзакций в соединении может быть изменен на неявные транзакции.

Неявные (имплицированные) транзакции. С момента переключения соединения на неявные транзакции все действия с БД выполняются в одной транзакции. Первая поступившая в соединении SQL-команда создает транзакцию, в которой далее выполняются все последующие операторы. За любым очередным оператором текущая транзакция может заканчиваться, а ее ре-

зультат фиксироваться или отменяться командами COMMIT и ROLLBACK. Команда COMMIT завершает транзакцию, сохраняя выполненные в ней изменения в БД. Команда ROLLBACK завершает (откатывает) транзакцию без изменений БД. Затем новая SQL-команда начнет следующую транзакцию, в которой выполняются все команды до фиксации или отката транзакции. Исполнение сервером транзакций в каждом соединении представляет глобальная переменная @@trancount — счетчик активных транзакций. При наличии транзакций @@trancount >= 1, в отсутствие @@trancount = 0. Пользователь может включать в выполняемую последовательность команд операторы фиксации или отката транзакции, которые действуют на изменения в БД, выполненные от предыдущей фиксации или отката.

Переключение между фоновыми режимами автоматических и неявных транзакций выполняется командой

SET IMPLICIT_TRANSACTIONS ON | OFF.

При задании параметра ON сервер переходит в режим неявных транзакций, OFF — устанавливает режим автоматических транзакций. Тип фоновых транзакций задается отдельно для каждого соединения с сервером.

Информацию о включенных опциях соединения возвращает команда

DBCC USEROPTIONS. Например, скрипт из двух команд:

SET IMPLICIT_TRANSACTIONS ON;

DBCC USEROPTIONS;

вернет в общем списке установленных опций IMPLICIT_TRANSACTIONS SET.

Серверная переменная @@OPTIONS в двоичном виде содержит информацию о параметрах соединения. Второй разряд @@OPTIONS хранит тип умалчиваемых транзакций. Выражение 2 & @@OPTIONS выделит значение второго разряда и вернет значение 0 или 2.

Рассмотрим пример исполнения заданий серверу в режиме автоматических транзакций:

SET IMPLICIT_TRANSACTIONS OFF;

PRINT 'Автоматические транзакции '+CAST (@@ TRANCOUNT AS VARCHAR (2));

— PRINT выводит @@TRANCOUNT = 0. Нет активных транзакций

SELECT * FROM Tab1;

PRINT ' Автомат.транзакции ' + CAST (@@TRANCOUNT AS VARCHAR (2));

— PRINT снова выводит 0 активных транзакций, т. к. запрос уже исполнен.

Теперь исполнение заданий серверу в режиме неявных транзакций.

В соединении устанавливается режим неявных транзакций:

SET IMPLICIT_TRANSACTIONS ON;

PRINT 'Неявные транзакции ' +CAST (@@TRANCOUNT AS VARCHAR (2));

— PRINT выводит 0, т. к. после переключения в режим еще не было SQL-команды (PRINT не SQL-команда, т. к. не работает с БД).

SELECT * FROM Tab1; — *SQL-команда, которая создает транзакцию;*

PRINT 'Неявные транзакции ' + CAST (@@TRANCOUNT AS VARCHAR (2));

— PRINT выводит значение переменной @@TRANCOUNT = 1.

COMMIT TRANSACTION — *заканчивает неявную транзакцию.*

PRINT 'Неявные транзакции ' + CAST (@@TRANCOUNT AS VARCHAR (2));

— PRINT выводит значение @@TRANCOUNT=0, т. к. транзакция зафиксирована

INSERT Tab1 VALUES (1, 'первая') — *SQL-команда начинает новую транзакцию*

PRINT 'Неявные транзакции ' + CAST (@@TRANCOUNT AS VARCHAR (2));

— *PRINT выводит @@TRANCOUNT = 1*

INSERT Tab1 VALUES (2, 'вторая') — *команда выполняется в уже созданной транзакции*

PRINT 'Неявные транзакции ' + CAST (@@TRANCOUNT AS VARCHAR (2));

— *PRINT выводит @@TRANCOUNT = 1*

SELECT * FROM Tab1 — *из Tab1 2 выводит строки*

ROLLBACK TRANSACTION — откатывает неявную транзакцию после чего @@TRANCOUNT = 0

SELECT * FROM Tab1 — *после отката в Tab1 нет строк, но select начинает новую транзакцию*

PRINT 'Неявные транзакции ' CAST (@@TRANCOUNT AS VARCHAR (2));

— *выводит @@TRANCOUNT = 1 активная транзакция, создана последним запросом*

На фоне автоматических и неявных транзакций могут быть заданы явные транзакции.

Явные транзакции определяются специальными командами Transact-SQL, отмечающими начало и конец явной транзакции. В состав явной транзакции не должны включаться команды создания, удаления и модификации базы (CREATE/DROP/ALTER DATABASE, …).

Начало явной транзакции выполняет команда

BEGIN TRAN [SACTION] [<*имя транзакции/имя переменной*> [WITH MARK ['<*метка, используемая при восстановлении БД по журналу транзакций*>']]].

Имя транзакции используется для идентификации вложенных транзакций. Имя может быть задано идентификатором или строковой переменной, в которой задано имя транзакции. В режиме неявных транзакция первая явная считается вложенной транзакцией.

Опция WITH MARK требует создания в журнале метки для транзакции. При наличии метки в процессе восстановления данных с помощью журнала база может быть восстановлена

в состояние, которое она имела к началу помеченной транзакции.

После объявления начала транзакции в ее составе могут использоваться любые операторы Transact-SQL, за исключением команд, создающих, изменяющих, копирующих и восстанавливающих базу данных. Кроме обычных SQL-команд, в явной транзакции может быть задана команда создания точки сохранения состояния базы SAVE TRAN [SACTION] *<имя точки сохранения/имя переменной>*.

Команда SAVE помечает состояние данных, к которому при необходимости можно будет вернуться путем выполнения отката базы на точку сохранения. Таким образом откат становится возможным не только к началу, но и к любому промежуточному состоянию БД в транзакции.

Окончание явной транзакции также отмечается командами COMMIT или ROLLBACK.

Команда COMMIT [TRAN [SACTION]] [*<имя транзакции>*/*<имя переменной>*]]; приводит к снятию блокировок данных и сохранению изменений, выполненных транзакцией, а также фиксирует факт успешного завершения указанной транзакции.

Команда ROLLBACK [TRAN [SACTION]] [*<имя транз./<имя перем.>/<имя точки сохранения>*]] отменяет в БД все или часть действий транзакции. Параметр оператора ROLLBACK определяет точку отката на состояние данных до начала транзакции или на момент создания точки сохранения.

Во вложенных транзакциях откат возможен только на начало внешней транзакции. Попытка отката к внутренней транзакции вызывает исключение и отмену исполнения команды. Уровень текущей вложенности в соединении активных транзакций хранит глобальная переменная @@TRANCOUNT. Каждая команда BEGIN [TRAN [SACTION]] увеличивает значения уровня вложенности транзакций @@TRANCOUNT на 1.

Каждая команда COMMIT [TRAN [SACTION]] уменьшает @@TRANCOUNT на 1. Но при этом фиксация изменений в БД

и освобождение заблокированных данных выполняется, только когда значение @@TRANCOUNT становится равным 0, т. е. только для внешней транзакции.

Команда SAVE TRAN [SACTION] не изменяет значение @@TRANCOUNT.

Пример использования вложенных фиксируемых транзакций:

BEGIN [TRAN [SACTION]]*<имя внешней тран.>*

— увеличивает @@TRANCOUNT на 1

.....

BEGIN [TRAN [SACTION]] <имя вложенной транзакции>

— увеличивает @@TRANCOUNT на 1

.....

COMMIT [TRAN [SACTION]] *<имя вложенной транзакции>*

— или COMMIT [TRAN [SACTION]] *<имя внешней транзакции>*

— или COMMIT [TRAN [SACTION] *<без имени>*

— уменьшают @@TRANCOUNT на 1

и если @@TRANCOUNT становится = 0 фиксирует внешнюю транзакцию

Пример отката вложенных транзакций.

BEGIN [TRAN [SACTION]]*<имя внешней транзакции>*

— увеличивает @@TRANCOUNT на 1

.....

BEGIN [TRAN [SACTION]] *<имя вложенной транзакции>*

— увеличивает @@ trancount

.....

ROLLBACK [TRAN [SACTION]] *<имя вложенной транзакции>*

— откат вложенной транзакции не предусмотрен, а значит, такая команда вызывает исключение и не изменяет @@ TRANCOUNT.

Использование отката ROLLBACK с указанием имени внешней транзакции

ROLLBACK [TRAN [SACTION]] *<имя внешней транзакции>*
или ROLLBACK — без имени транзакции
откатывает все вложенные транзакции, возвращает базу к исходному состоянию и устанавливает @@TRANCOUNT = 0.

Команда ROLLBACK [TRAN [SACTION]] *<имя точки сохранения>* откатывает БД к состоянию на момент создания точки сохранения, не изменяя значения @@TRANCOUNT. При использовании вложенных транзакций результат в БД (откат или фиксация) определяется только по отношению к внешней транзакции. Такие же правила обработки вложенных транзакций действуют, когда в составе явной транзакции выполняется вызов программного модуля (например, хранимой процедуры), в котором используется своя транзакция. Транзакция в процедуре окажется вложенной по отношению к вызвавшей процедуру транзакции.

Рассмотрим пример изменения состояния данных с использованием вложенных транзакций в одном соединении. Доступность добавленных или измененных в транзакции данных из других соединений зависит от типа используемых блокировок и установленных уровней изоляции транзакций, которые будут рассмотрены далее.

Пусть исходно в базе имеется пустая таблица Tab1.

BEGIN TRANSACTION T1

INSERT Tab1 VALUES (1, 'Первая') — *добавление первой записи*

SAVE TRANSACTION P1 — *создание точки сохранения после включения 1-й записи*

INSERT Tab1 VALUES (2, 'Вторая') — *добавление второй записи во внешней транзакции*

BEGIN TRANSACTION T2 — *создание вложенной транзакции. Две активных транзакции @@TRANCOUNT=2*

INSERT Tab1 VALUES (3, 'Третья') — *добавление записи во вложенной транзакции*

SELECT * FROM Tab1 — *будет показано 3 записи. При этом доступность добавленных записей в другом соединении зависит от установленного уровня изоляции транзакций*

ROLLBACK TRANSACTION P1 — *откат на точку сохранения не изменяет числа активных транзакций. Осталось 2 активных транзакции*

SELECT * FROM Tab1 — *после отката в таблице осталась 1 запись и 2 активных транзакции*

COMMIT TRANSACTION T2

SELECT * FROM Tab1 — *в таблице 1 запись и 1 активная транзакция*

ROLLBACK TRANSACTION — *в соединении не осталось активных транзакций*

SELECT * FROM Tab1 — *таблица Tab1 опять пуста, так как выполнен откат внешней транзакции*

Рассмотрим пример применения явной транзакции для проведения в базе гарантированно целостных изменений данных, касающихся нескольких таблиц. Пусть в базе печатных изданий требуется изменение первичного ключа книги (PK) со старым значением, находящимся в переменной @ID_OLD, на значение из переменной @ID_NEW. Для сохранения всех связей записей (см. рис. 2.1*)* эти изменения должны быть выполнены в трех таблицах.

BEGIN TRANSACTION — *начало явной транзакции без имени*

ALTER TABLE Titleauthor CONSTRAIN FK_Titles_Titleauthor NOCHECK

— *отключение проверки ограничения внешнего ключа (FK_Titles_Titleauthor) в таблице Titleauthor*

ALTER TABLE Sales … — *аналогичное отключение проверки ограничения FK в таблице Sales*

— *изменение первичного ключа в Titles*

UPDATE Titles SET Title_ID = @ID_NEW Where Title_ID = @ID_OLD

— *изменение внешнего ключа в Titleauthor*

UPDATE Titleauthor SET Title_ID = @ID_NEW Where Title_ID = @ID_OLD

— *аналогичное изменение внешнего ключа Title_ID в таблице Sales*

UPDATE Sales **…**

— восстановление проверки ограничения FK таблицы Titleauthor

ALTER TABLE Titleauthor constrain FK_Titles_Titleauthor CHECK

… — аналогичное восстановление проверки ограничения FK таблицы Sales ALTER TABLE Sales

COMMIT TRANSACTION — *фиксация явной транзакции*

При использовании явной транзакции ошибка при выполнении какой-либо команды или отказ сервера не приводит к разрушению связей между записями из обрабатываемых таблиц.

Использование транзакции для программирования обработки данных

Возможность отмены изменений данных путем отката транзакций создает дополнительные возможности построения программ обработки данных, при котором действия сначала выполняются, а потом в зависимости от результата сохраняются или отменяются по следующей схеме:

- начало транзакции,
- выполнение действий в БД,
- проверка условия (состояния) данных в базе,
- фиксация транзакции (сохранение результата) при выполнении некоторого условия,
- откат транзакции при невыполнении в БД проверяемого условия.

Обработка ошибок в явной транзакции

Реакция сервера на ошибку в теле транзакции зависит от значения опции соединения XACT_ABORT.

При задании SET XACT_ABORT OFF ошибка в каком-либо операторе в составе транзакции отменяет выполнение только этого оператора, но не откатывает всю транзакцию. Если SET XACT_ABORT ON, то при любой ошибке сервер откатывает всю (внешнюю) транзакцию. Состояние опции XACT_ABORT возвращает команда DBCC USEROPTIONS.

Пример влияния ошибок на исполнение в транзакции.

— *Создание тестовой таблицы tab1*

```
WHILE EXISTS (SELECT * FROM SYS. TABLES WHERE Name
= 'tab1')
DROP TABLE tab1;
CREATE TABLE tab1 (F1 INT NOT NULL PRIMARY KEY);
GO;
```

Выполнение транзакций при установке отмены только ошибочного оператора (SET XACT_ABORT OFF) без отката транзакции

```
BEGIN TRANSACTION tt1
  INSERT INTO tab1 VALUES (1); — @@TRANCOUNT = 1
  BEGIN TRANSACTION tt2
INSERT INTO tab1 VALUES (2); — @@TRANCOUNT = 2
ROLLBACK TRANSACTION tt2
```

— *Выводит сообщение*: Cannot roll back tt2. No transaction or savepoint of that name was found. (Не выполнен откат вложенной транзакции. Только этот оператор транзакции отменен).

```
— Выполнение транзакции продолжается
INSERT INTO tab1 VALUES (3); — @@TRANCOUNT = 2
COMMIT TRANSACTION tt1 — Фиксация транзакции @@
TRANCOUNT = 1
COMMIT TRANSACTION — Фиксация внешней тран. @@
TRANCOUNT = 0
GO;
SELECT * FROM tab1; — выводит три строки из таблицы tab1
```

Пример выполнения транзакций при указании отмены всех изменений при возникновении ошибки (SET XACT_ABORT ON)

```
SET XACT_ABORT ON; — откат внешней транзакции при
возникновении исключения
BEGIN TRANSACTION tt1;
INSERT INTO tab1 VALUES (1); — @@TRANCOUNT = 1
BEGIN TRANSACTION tt2
INSERT INTO tab1 VALUES (2); — @@TRANCOUNT = 2
```

SELECT * FROM tab1; — *в таблице 2 строки.*

ROLLBACK TRANSACTION tt2 — *ОШИБОЧНЫЙ оператор*

— Сервер выводит: Cannot roll back tt2. No transaction or savepoint of that name was found.

Происходит автоматический откат внешней транзакции. Остальные команды до конца пакета (до GO) отменяются, не изменяя БД).

INSERT INTO tab1 VALUES (3);

COMMIT TRANSACTION tt1

PRINT Cast (@@TRANCOUNT As CHAR (3));

COMMIT TRANSACTION tt1

PRINT '@@TRANCOUNT = ' + Cast (@@TRANCOUNT As CHAR (3))

GO;

PRINT '!!! @@TRANCOUNT = ' + Cast (@@TRANCOUNT As CHAR (3))

— *выводит @@TRANCOUNT = 0*

SELECT * FROM tab1; — *выводит пустую таблицу tab1*

Обработка исключений в явных транзакциях

Ошибки, приводящие к исключению внутри явной транзакции, приводят транзакцию к состоянию нефиксируемой. Поэтому оператор COMMIT не может использоваться в блоке CATCH. Допустимо применение только команды ROLLBACK.

Например:

SET XACT_ABORT OFF; — *отменяет только ошибочный оператор*

BEGIN TRY

BEGIN TRANSACTION T1

INSERT Tab1 VALUES (1, 'Первая')

SELECT * FROM Tab1 — *в таблице 1 запись и 1 активная транзакция*

INSERT Tab1 VALUES ('Ошибка ', 'Вторая') — *ошибка в типе данного для первого поля вызывает исключение. Эта запись не добавляется в таблицу*

COMMIT TRANSACTION T1
END TRY
BEGIN CATCH — *блок обработки исключений*
— *вывод сообщений об ошибке*
SELECT ERROR_NUMBER () As ErrorNumber, — *выводит код ошибки 245*
ERROR_MESSAGE () As ErrorMessage; — *выводит Conversion failed…*
COMMIT TRANSACTION T1 — *количество активных транзакций 1, попытка ее фиксации приводит к ошибке с кодом 3930 –текущая транзакция не может быть зафиксирована. В блоке CATCH. Следовало использовать команду* ROLLBACK TRANSACTION T1.

END CATCH

Для анализа состояния транзакции предназначена функция XACT_STATE ().

Функция XACT_STATE () возвращает значения:

1 — транзакция фиксируема, 0 — нет активной транзакции, —1 — транзакция нефиксируема и может быть только отменена. Для нефиксируемой транзакции команда COMMIT вызывает исключение.

Контрольные вопросы по теме транзакции базы данных:

1. К чему приведет выполнение таких SQL-команд?

BEGIN TRANSACTION
CREATE DATABASE NEW
COMMIT TRANSACTION

2. Что выполнит следующий скрипт?

BEGIN TRANSACTION
WHILE NOT EXISTS (SELECT * FROM sys.tables WHERE Name = 'tab1')
CREATE TABLE new (T1 int)
COMMIT TRANSACTION

1.2. Блокировки и версионность данных

Для независимого совместного выполнения транзакций в режиме оперативной обработки данных (OLTP) в серверах баз данных реализованы механизмы выполняющие:

— защиту (блокировку) обрабатываемых транзакцией данных,

— создание транзакцией дополнительных временных версий данных.

Блокировка — ограничение доступа к данным, обрабатываемым пользовательской транзакцией для других параллельно выполняемых транзакций. Это основной метод защиты, используемый в серверах баз данных.

Блокировки данных реализуются менеджером блокировок, который устанавливает, анализирует, снимает блокировки данных. Типы и виды применяемых блокировок и способы их использования определяются возможным влиянием одних транзакций на результаты, создаваемые в параллельно исполняемых транзакциях.

Выделены четыре различных типа взаимного влияния одной транзакции на другую. Их принято называть «проблемы параллельной обработки транзакций».

Проблемы параллельной обработки транзакций

1. Проблема последнего обновления (потери обновления) возникает при попытках двух и более транзакции обновить одни и те же данные. Возможное влияние параллельных транзакций, изменяющих запись R, показана на рис. 1.3.

Транзакция T1 в момент t1 д читает запись R для внесения изменения. В момент t_2 эту же запись, также для изменений читает транзакция T2. В момент t_3 транзакция T1 сохраняет измененную запись в базе, которая в последующий момент t_4 будет снова изменена транзакцией T2. Таким образом, изменения, выполненные транзакцией T1, будут потеряны (заменены транзакцией T2).

Рис. 1.3. Потеря изменений данных, выполняемых двумя параллельными транзакциями

2. Проблема «грязного» чтения.

Пусть R1, R2 — записи, хранящие суммы средств на счетах клиентов банка. Параллельно выполняются две транзакции:

— T1 — передача суммы со счета R1 на счет R2,

— T2 — расчет общей суммы средств на счетах клиентов.

Возможная последовательность событий обработки данных транзакциям приведена на рис. 1.4.

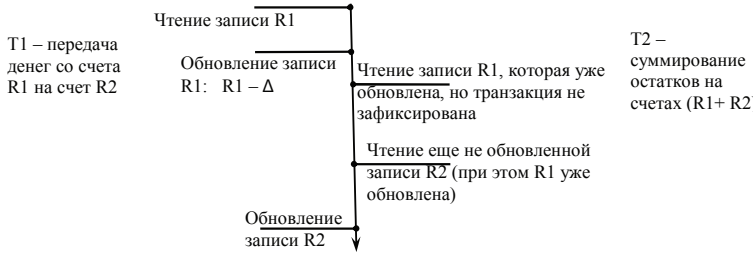


Рис. 1.4. Пример «грязного» чтения

Если транзакция T1 прочитает и изменит первую запись R1, а транзакция T2 прочитает ее до изменения значения второго слагаемого в записи R2 и фиксации изменений, то сумма, формируемая транзакцией T2, будет содержать ошибку, равную величине суммы, передаваемой со счета R1 на счет R2.

Ошибка суммы будет при чтении незафиксированных изменений и в случае отката T1.

3. Проблема неповторяемого чтения.

Транзакции T1 необходимо прочитать запись R, например, чтобы сначала по этим данным рассчитать значение, которое в следующем запросе используется в условии фильтрации транзакции T1. При этом перед исполнением запроса параллельная транзакция T2 успевает обновить запись R (рис. 1.5). При выполнении запроса будет использовано условие, не соответствующее текущим данным.

Рис. 1.5. Неповторяемое чтение в транзакции T1

4. Проблема фантомов.

Фантом — транзакция получает данные, не существующие в базе. Пусть транзакция T1 добавляет запись R в таблицу, а транзакция T2 выводит данные из этой таблицы, как показано на рис. 1.6. Если после добавления записи R транзакцией T1, другая транзакция T2 ее прочитала, а затем был выполнен откат транзакции T1, то в транзакции T2 будет получена несуществующая запись.

Основной способ разрешения конфликтов параллельно исполняемых транзакций — установка запрета доступа к данным (блокировки данных) для других транзакций еще до начала выполнения операций данной транзакции [2]. Так как блокировки снижают производительность сервера, важно ограничивать время и количество блокируемой информации.

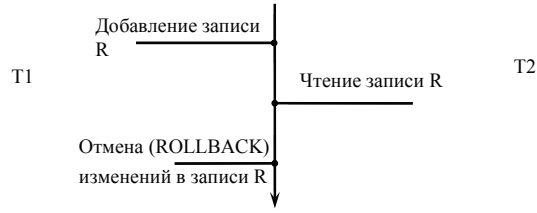


Рис. 1.6. Пример возникновения фантома

В MS SQL SERVER и других СУБД реализованы различные типы блокировок, отличающиеся возможностью совместного доступа к данным:

- Разделяемая блокировка (S — Shared),
- Монопольная блокировка (X — eXclusive),
- Блокировка обновлений (U),
- Блокировка диапазона ключей (K — Key Lock),
- Блокировка схемы (SL — Scheme Lock),
- Разделяемая блокировка намерений (IS — Intent Shared),
- Монопольная блокировка намерений (IX — Intent eXclusive).

Разделяемая блокировка (S — Shared) устанавливается транзакцией, выполняющей чтение данных (команды SELECT, FETCH).

Разделяемая блокировка разрешает установку новых разделяемых блокировок и чтение этих же данных другими транзакциями, но запрещает доступ к данным транзакциям, требующим установку блокировок другого типа.

Таким образом, данные могут одновременно считываться многими транзакциями, установившими блокировки типа S. Заканчивая работу, транзакция снимает свою блокировку и открывает доступ к данным другим транзакциям. Например,

<Читаемые данные> ← две S-блокировки читающих транзакций T1–S и T2 — S.

Монопольная блокировка (X — eXclusive) не может быть установлена на заблокированные данные, т. е. те данные,

на которые уже установлены блокировки типа S или X. Однако, будучи установленной монопольная блокировка запрещает установку других блокировок и защищает данные от чтения, удаления и изменения. Монопольная блокировка необходима транзакциям, содержащим команды обновления и удаления данных (Update, Delete).

<Изменяемые данные> ← X-блокировка установлена транзакцией T.

Блокировка обновлений (U) занимает промежуточное положение между разделяемым и монопольным типами. Блокировка обновлений заменяет ранее установленные разделяемые блокировки (S → U) в том случае, если новая транзакция пытается установить монопольную блокировку этих данных. Заменив блокировки S, блокировка U разрешает чтение данных транзакциями, ранее установившими свои разделяемые блокировки, но запрещает установку новых разделяемых блокировок. Таким образом обеспечивается обработка транзакций в порядке их поступления на сервер.

Блокировка схемы (SL — Schema Lock) защищает структуру объекта базы данных от внесения изменений. Блокировка схемы устанавливается на строки системных таблиц словаря базы при выполнении DML, DDL и DCL-операторов. При этом данный вид блокировки устанавливается как на сам объект, так и на все от него зависящие. Например, блокировка схемы для таблицы вызовет аналогичную блокировку представлений, которые используют данную таблицу. Блокировка SL несовместима ни с каким другим типом блокировок. Блокировка схемы выполняется сервером автоматически на время выполнения транзакции.

Блокировка диапазона ключей (K — Key Lock). Ранее рассмотренные блокировки защищали существующие данные, но не могли защитить таблицы базы от появления новых строк. Особенность данной блокировки в том, что блокируется доступ не к строкам таблицы, а к индексу, построенному

по первичному ключу. Таким образом, запрещается создавать в индексе новые ключи, а следовательно, и в таблицу не могут быть добавлены новые строки.

Блокировки намерений (Intent) имеют два типа:

– разделяемая (IS — Intent Shared),
– монопольная (IX — Intent eXclusive).

Сервер устанавливает блокировку намерений на не непосредственно обрабатываемые данные, а на данные, прогнозируемые для последующей обработки. Такая блокировка позволяет избежать блокировок типа X или S, которые могут быть установлены другими транзакциями на последующие обрабатываемые данные. Опережающие блокировки намерений устанавливаются сервером автоматически на основе анализа множества обрабатываемых записей.

Кроме типа, при установке блокировок задается уровень блокировки, определяющий количество одновременно блокируемых данных.

В MS SQL SERVER менеджер блокировок поддерживает следующие уровни блокировок:

– строка таблицы,
– страница данных на сервере (8 Кб),
– экстент (8 страниц выделяемой дисковой памяти),
– отдельная таблица,
– вся база данных.

В параметрах управления блокировками могут быть установлены блокировки строк, страниц, таблицы или всей базы. Без специальных указаний в параметрах SQL-команд сервер самостоятельно выбирает уровень устанавливаемой блокировки. По мере выполнения SQL-команды уровень блокировки может автоматически изменяться, обычно в сторону возрастания. Например, переходя от блокирования множества страниц к блокированию всей таблицы.

Информацию о текущих блокировках возвращает представление SYS.DM_TRAN_LOCKS. Следующий сценарий демонстри-

рует блокировки, устанавливаемые сервером автоматически при добавлении записи в таблицу authors. Глобальная переменная @@SPID содержит идентификатор текущего соединения:

use pubs

Begin transaction

INSERT authors VALUES ('111–11–0004', 'Юрий', '223 226–8884', 'Петров', 'Москва', 'РФ', '94111', 1);

SELECT resource_type, request_mode, request_type

FROM SYS.DM_TRAN_LOCKS WHERE request_session_id = @@SPID;

Результат запроса представлен в табл. 1.

Таблица 1

Текущие блокировки в транзакции

Resource_type	Eqest_mode	Request_type
DATABASE	S	LOCK
PAGE	IX	LOCK
OBJECT	X	LOCK
KEY	X	LOCK
KEY	X	LOCK
KEY	X	LOCK
KEY	X	LOCK

«Мертвые» блокировки — клинчи транзакций (Dead lock)

Одновременное выполнение многих блокирующих данные транзакций может создать ситуации, при которых транзакции взаимно блокируют необходимые им данные.

Как показано на рис. 1.7, транзакция T1 монопольно заблокировала набор записей A и обратилась для чтения к записям B, блокированным транзакцией T2. Если при этом для транзакции T2 потребуются записи A, то обе транзакции не смогут выполняться в ожидании снятия блокировок.

Способы снятия «мертвых» блокировок

Первый способ — выполняемый сервером откат одной из заблокированных транзакций. Менеджер блокировок че-

рез определенные интервалы времени проводит анализ заблокированных данных. Обнаружив цепочку взаимно блокирующих транзакций, сервер разрешает их конфликт путем отката одной из транзакций. При этом в соединении с откатываемой транзакцией посылается сообщение об ошибке с кодом 1205. В приложении, запустившем откатываемую транзакцию, должна быть предусмотрена обработка такой ошибки. После отката другие транзакции получают доступ к разблокированному ресурсу и продолжают выполнение. Для управления откатываемой транзакцией в случае возникновения «мертвых» блокировок им могут быть назначены приоритеты. Задание транзакции приоритета выполняет оператор SET DEADLOCK_PRIORITY Low | Normal | *<@имя переменной>*.

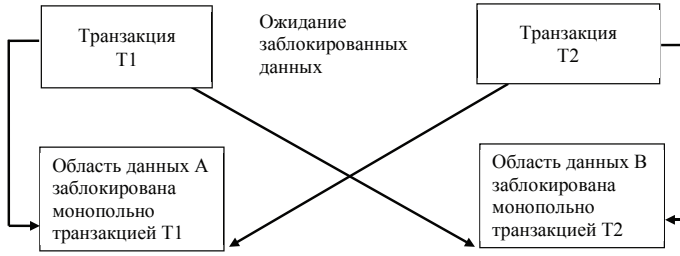


Рис. 1.7. Взаимные блокировки двух транзакций

Исходно все транзакции имеют приоритет Normal. Применив в соединении данный оператор с параметром Low, можно понизить приоритет исполняемых транзакций. Пониженное значение приоритета (Low) может быть задано значением строковой переменной.

Второй способ снятия «мертвых» блокировок реализуется установкой ограничения на время ожидания транзакцией разблокирования требуемого ресурса. Для этого используется оператор SET LOCK_TIMEOUT *<время ожидания в миллисекундах>*.

По истечении заданного времени транзакция откатывается с генерацией исключения. Время ожидания исполнения транзакции устанавливается в соединении и действует на все транзакции этого соединения.

Для профилактики «мертвых» блокировок можно использовать следующие рекомендации:

- использовать в приложениях кеш прочитанных из базы данных (например, используя разделенные классы технологии ADO. NET);
- создавать транзакции с минимальным объемом кода. Для этого на диалоговых формах приложений для пользователей БД предусмотреть кнопки «Сохранить», «Изменить», «Добавить», при нажатии которых на сервер посылаются короткие обновляющие транзакции. В остальное время работы пользователя данные не блокируются;
- устанавливать в обновляющих транзакциях минимальный необходимый уровень блокировки данных;
- если есть возможность, выполнять обработку данных в разных транзакциях в одном логическом порядке (просмотр таблицы в одном направлении);
- обходиться транзакциями, однократно читающими одни данные.

Средства управления блокировками данных в обрабатывающих транзакциях

В серверах БД предусмотрены два механизма управления блокировками.

1. Установка в соединении с сервером БД определенного уровня изоляции своих транзакций. В соответствии со стандартом SQL/92 серверы поддерживают четыре уровня изоляции транзакций по числу типов проблем, возникающих при совместной обработке данных. Каждый последующий уровень устраняет еще одну проблему параллельной обработки, увеличивая степень изоляции транзакций. При этом более высокие

уровни изоляции снижают возможности параллельной обработки данных, а значит, потенциально снижают производительность сервера.

2. Установка индивидуальной блокировки данных, используемых конкретной SQL-командой внутри транзакции.

Уровни изоляции транзакций

Уровень изоляции устанавливается в соединении с сервером и действует на все последующие транзакции до смены уровня или разрыва соединения. После того как соединение закрывается и возвращается в пул, сохраняется уровень изоляции последней инструкции SET TRANSACTION ISOLATION LEVEL. Поэтому в новых подключениях, повторно использующих ресурсы пула соединений, применяется уровень изоляции, который действовал в момент возврата соединения в пул.

Установка уровня изоляции транзакций выполняется оператором

SET TRANSACTION ISOLATION LEVEL <*уровень изоляции*>.

Если в соединении вызывается хранимая процедура или триггер, изменяющие уровень изоляции, то после выполнения процедуры в точке возврата автоматически восстанавливается ранее использовавшийся уровень.

Уровни изоляции транзакций в MS SQL SERVER

Различные стандартные уровни изоляции реализуются сервером путем установки и снятия с обрабатываемых записей соответствующих блокировок. Блокировка может быть установлена только на время доступа к записи или сохраняться до конца транзакции.

Наименьшую защиту обрабатываемых в транзакциях данных обеспечивает уровень READ UNCOMMITTED — чтение незафиксированных изменений (уровень 0 в стандарте SQL и уровень 1 в MS SQL SERVER). На этом уровне с помощью блокировок поддерживаются:

– запрет обновления одних и тех же данных параллельными транзакциями;

– данные, обновленные транзакцией, до ее окончания не доступны для обновления транзакциям из других соединений, но доступны для чтения другими транзакциями.

Уровень READ UNCOMMITTED исключает проблему последнего обновления, но не защищает от других проблем: «грязное» и неповторяемое чтение и фантомы.

READ COMMITTED — чтение зафиксированных изменений (уровень 1 в Microsoft Server или 2 — в стандарте SQL).

Сервер снимает блокировку сразу после чтения данных, а значит, допускает их изменение другой транзакцией, но сохраняет монопольные блокировки измененных данных до конца транзакции. Следовательно, запрещает и изменение и чтение обновленных данных до окончания изменяющей транзакции. Этот уровень устанавливается в новом соединении с MS SQL SERVER по умолчанию. При использовании READ COMMITTED данные защищены от проблем последнего обновления и «грязного» чтения. Действующий по умолчанию уровень READ COMMITTED является основным для обычной работы в приложениях, обрабатывающих исключения, вызванные ошибками при совместной обработке данных.

REPEATABLE READ — повторяемое чтение (уровень 2 для Microsoft Server и 3 — в стандарте SQL). Любая транзакция сохраняет блокировку всех обновляемых и читаемых данных до конца транзакции. В дополнение к двум предыдущим уровень REPEATABLE READ исключает проблему неповторяемого чтения, но сохраняется возможность появления фантомов.

SERIALIZABLE — серийное чтение (наивысший уровень изоляции — 3 или 4 в стандарте SQL). В дополнении к предыдущему уровню исключается появление фантомов за счет блокировки диапазона ключей в кластерном индексе. Запрет на изменение множества ключей защищает от удаления строк и добавления новых строк в таблицы.

Влияние уровней изоляции транзакций на защиту данных представлено в табл. 2.

Таблица 2

Защита данных для разных уровней изоляции транзакций

Ключевые слова	**«Грязное» чтение —** чтение данных, измененных в др. незафиксированной транзакцией	**Неповторяемое чтение** — чтение данных, измененных в другой зафиксированной или своей не зафиксированной транзакцией	**Чтение фантомов —** например, чтение, изменение, снова чтение одних данных в одной транзакции
READ UNCOMMITTED — чтение незафиксированных изменений	Возможно, так как при чтении не проверяется X-блокировка (X меняется на S)	Возможно	Возможно
READ COMMITTED — чтение зафиксированных изменений	Исключено, так как запрещено чтение монопольно заблокированных данных	Возможно, если выполняется повторное чтение, но данные были изменены другой зафиксированной транзакцией	Возможно
REPEATABLE READ — повторяемое чтение	Исключено, так как запрещено чтение монопольно заблокированных данных	Исключено, так как сохраняет разделяемую блокировку прочитанных данных до конца транзакции	Возможно. За счет добавления и удаления строк фиксированной транзакцией
SERIALIZABLE — серийное чтение	Исключено, так как запрещено чтение монопольно заблокированных данных	Исключено, так как сохраняет разделяемую блокировку прочитанных данных до конца транзакции	Исключено, так как блокируется диапазон ключей обрабатываемых строк

Информация о текущих параметрах соединений содержится в представлении sys.dm_exec_sessions. Для каждого соединения

имеется отдельная строка. Действующий уровень изоляции хранится целочисленным кодом в столбце transaction_isolation_level. Для обозначения уровня изоляции используются следующие коды: 0 — Неопределенный, 1 — Read Uncomitted, 2 — Read Committed, 3 — Repeatable Read, 4 — Serializable, 5 — Snapshot (используется версионность данных). Например,

Запрос: SELECT transaction_isolation_level FROM sys.dm_exec_sessions

WHERE session_id = @@SPID;

вернет для transaction_isolation_level значение 2.

Индивидуальная блокировка данных

На данные каждой таблицы, обрабатываемые отдельной командой, могут быть установлены индивидуальные блокировки. Установленная в команде блокировка обычно снимается после ее выполнения, но может быть продлена до конца транзакции.

Индивидуальная блокировка в SQL-команде задается в предложении FROM за именем таблицы в виде подсказки:

FROM… <*имя таблицы*> WITH (<*способ блокировки данных в таблице*>)…

Требуемый способ блокировки задается набором соответствующих ключевых слов. Для каждой используемой в команде таблицы может быть указан свой набор ключевых слов, задающих тип (S или X) и уровень (строка, страница, таблица) блокирования.

Ключевые слова, задающие уровень разделяемых блокировок (S):

- RowLock — устанавливает разделяемую построчную S-блокировку на таблицу,
- PagLock — постраничная S-блокировка,
- TabLock — S-блокировка всей таблицы.

Например, оператор Select… FROM authors WITH (TabLock) — блокирует все строки таблицы, разрешая чтение и запрещая изменения данных другими транзакциями.

С ключевыми словами для S-блокировки (RowLock, PagLock, TabLock) допустима уточняющая опция блокировки UpdLock. При использовании UpdLock в случае появления другой транзакции, требующей монопольной блокировки этих данных, сервер заменит разделяемую блокировку на блокировку обновлений (S → U).

Ключевые слова для установки (подсказки) монопольной блокировки (X):

– XLock — в сочетании с ключевыми словами RowLock (блокировка строк) или PagLock (постраничная блокировка) устанавливает для этих данных монопольную блокировку;
– TabLockX — требует установки монопольной блокировки всей таблицы. TabLockX совместима только с уровнем TabLock.

Заданные в параметре FROM блокировки по умолчанию действуют на данные, обрабатываемые только этим SQL-оператором. Но действие индивидуальных блокировок для таблицы можно распространить до конца транзакции. Для этого используется опция задержки блокировки — HoldLock. Для команд, не имеющих подсказок блокировки, используются блокировки, определяемые установленным уровнем изоляции.

Например, задание в запросе таблицы authors в виде FROM authors WITH (TABLOCK, HOLDLOCK) разделяемо блокирует таблицу до конца транзакции, т. е. команд COMMIT или ROLLBACK.

Индивидуальные блокировки действуют на фоне установленного в соединении уровня изоляции и, как правило, блокировки отдельных таблиц используются для повышения защиты данных по сравнению с фоновым уровнем изоляции. Однако если в целом установлен высокий уровень изоляции, то в отдельном SQL-операторе можно отменить блокировку данных подсказкой NoLock (отмена блокировки).

В параметре FROM WITH (…) для отдельной таблицы в рамках одного SQL-оператора или в сочетании с HoldLock

в целом для транзакции могут быть использованы подсказки, эквивалентные соответствующим уровням изоляции: ReadUncommitted, ReadCommitted, RepeatableRead, Serializable. Этими ключевыми словами задаются соответствующие уровни изоляции, но применительно к определенной таблице.

Защита данных с помощью мгновенных снимков (версионности) — SNAPSHOT

Другой метод разрешения конфликтов параллельной обработки БД — создание транзакциями версий данных.

При включении режима версионности транзакция порождает копию (версию) обрабатываемых данных. При этом блокировки на обрабатываемые данные не устанавливаются, а значит, параллельные транзакции не ждут разблокирования данных. Созданная и, возможно, измененная в транзакции версия данных другим транзакциям не видна, до тех пор, пока эта транзакция не зафиксируется. Использование версий создает благоприятные условия для читающих транзакций:

- каждая читающая команда получает из БД последнюю зафиксированную версию записи;
- изменения данных в версии недоступны другим транзакциям до фиксации транзакции;
- другие транзакции получают все измененные данные в окончательном состоянии на момент фиксации транзакции.

Однако при фиксации транзакции обновляющей строки, которые были изменены другой транзакцией уже после создания снимка данных, возникнет конфликт версий. Конфликты версий данных разрешаются генерацией исключений, которые приводят к откатам транзакций. Так как откаты транзакций являются ресурсоемкими операциями, версионность используется при маловероятных конфликтах обновлений.

Использование мгновенных снимков в MS SQL SERVER

Применение мгновенных снимков данных требует включения специального разрешения для базы. По умолчанию верси-

онность включена только для служебных БД MASTER и MSDB и не может быть отключена.

Для пользовательских баз необходима предварительная установка режима мгновенных снимков командой, изменяющей параметры базы:

ALTER DATABASE <*имя БД*> SET ALLOW_SNAPSHOT_ISOLATION ON/OFF.

Значение ON включает механизм мгновенных снимков в БД.

После включения изоляции моментального снимка обновленные версии строк для каждой транзакции сохраняются в TEMPDB. Для работы с моментальными снимками данных каждая транзакция имеет уникальный порядковый номер. Эти уникальные номера записываются в каждой версии строки. Таким образом, для строки поддерживается несколько версий. Транзакция работает с последними версиями строк, имеющими порядковый номер, предшествующий порядковому номеру исполняемой транзакции, а значит, использует данные, существовавшие на момент ее начала. Более новые версии строк, созданные после начала транзакции, не доступны в этой транзакции. При этом, если транзакция моментального снимка попытается зафиксировать обновление строки, измененной после ее запуска, то возникнет ошибка, которая приведет к откату транзакции.

В режиме создания мгновенных снимков к четырем основным уровням изоляции добавляется уровень SNAPSHOT. Установку уровня выполняет команда SET TRANSACTION ISOLATION LEVEL SNAPSHOT.

Для основного уровня изоляции транзакций (READ_COMMITTED) в MS SQL SERVER использование мгновенных снимков устанавливается отдельным параметром базы READ_COMMITTED_SNAPSHOT ON, который задается командой ALTER DATABASE <*Имя БД*> SET READ_COMMITTED_SNAPSHOT ON. Если параметр базы READ_COMMITTED_SNAPSHOT находится в состоянии ON, для защиты данных

используется механизм управления версиями строк независимо от значения параметра ALLOW_SNAPSHOT_ISOLATION. Если параметр READ_COMMITTED_SNAPSHOT в состоянии OFF, при выполнении операций чтения текущей транзакцией сервер для предотвращения изменения строк другими транзакциями используют разделяемые блокировки.

Значения параметров, разрешающих использование версионности, могут быть получены через системное представление SELECT * FROM SYS. DATABASES.

Часть столбцов из SYS. DATABASES, содержащих свойства баз данных, показана в табл. 3.

Таблица 3

Системная информация о базах, использующих версионность

Name	Snapshot_isolation_state	Snapshot_isolation_state_desc	Is_read_committed_Snapshot_on
Master	1	ON	0
Tempdb	0	OFF	0
Model	0	OFF	0
Msdb	1	ON	0
ReportServer	0	OFF	0
Test	1	ON	1

Из табл. 3 следует, что базы MASTER, MSDB и Test используют версионность. Кроме того, в базе Test операция чтения основана на просмотре моментальных снимков и не запрашивает блокировки данных.

1.3. Средства для разработки серверной компоненты

1.3.1. Курсоры баз данных

Обычные DML-команды языка SQL основаны на реляционной алгебре, оперирующей множествами кортежей, и поэтому не имеют средств для доступа к определенному кортежу в от-

ношении. Однако в обработке БД существует ряд задач, требующих построчной обработки таблиц. Для этой цели в стандарт SQL включены курсоры. Курсор — программная конструкция, а не самостоятельный объект базы, обеспечивающая построчный доступ к данным, определяемым оператором SELECT [3]. Курсор содержит набор строк, в котором одномоментно доступна одна строка, называемая текущей. Установка текущей строки выполняется командами перехода — указания следующей обрабатываемой строки. Поля текущей строки читаются в переменных программах и доступны для анализа. Курсоры предназначены для создания программ, в которых требуется просмотр строк результата запроса и выполнения различных задач обработки данных в зависимости от значений полей текущей строки. Последовательным или прямым переходам по строкам обеспечивается обработка всего курсора. Общая схема использования курсора показана на рис. 1.8.

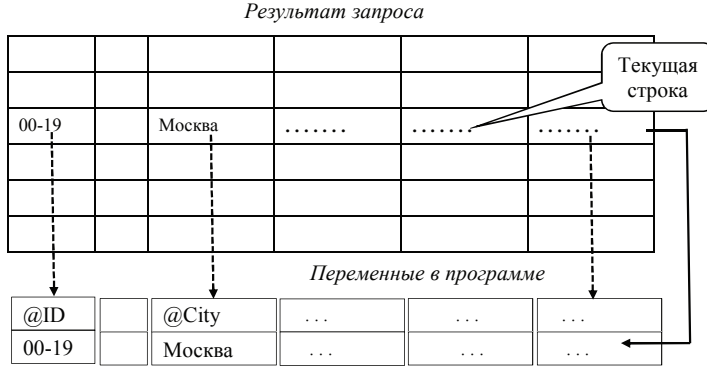


Рис. 1.8. Схема работы курсора

В клиент-серверных системах различают курсоры клиента и сервера.

На стороне клиента курсоры создаются средствами разработки приложений для баз данных. Способы разработки

Windows-приложений с созданием курсоров рассмотрены в пособии [4].

В данном разделе изучаются способы создания и обработки курсоров на стороне MS SQL SERVER. Курсоры сервера могут создаваться и использоваться внутри пакета операторов, хранимой в базе процедуры, функции или триггера на Transact-SQL.

Курсоры в MS SQL SERVER могут не только читать данные, но и вносить изменения в базу через присваивания новых значений в текущую строку. Курсоры, используемые для изменения и удаления данных, накладывают определенные ограничения на оператор SELECT, образующий набор строк в курсоре.

В SQL SERVER курсоры различаются по способу выбора текущей строки:

– последовательные курсоры (FORWARD ONLY), которые обеспечивают однократный перебор массива строк от начала к концу;
– прокручиваемые курсоры (SCROLL), обеспечивающие последовательный доступ от начала к концу и обратно, а также прямой переход к строке по ее номеру.

По способу создания и использования курсоры бывают следующих типов:

– статический курсор (STATIC). В момент активизации (открытия) курсора выполняется команда SELECT, осуществляющая выборку строк с последующей их записью в базу TEMPDB. Доступ к строкам статического курсора выполняется в копии данных, поэтому любые изменения базы после создания курсора не отображаются в его значениях. Через статические курсоры также не могут быть внесены изменения в БД. Они всегда открываются только для чтения (READ ONLY);
– ключевой курсор (KEYSET). При открытии курсора из БД извлекаются значения ключей тех строк таблиц, которые используются в создании курсора. В системной базе

TEMPDB создаются наборы ключей обрабатываемых строк. Пример построения ключей для связанных запросом строк из двух таблиц показан на рис. 1.9.

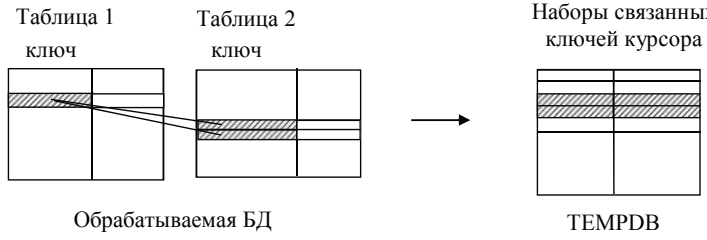


Рис. 1.9. Схема создания набора связанных ключей курсора

При переходе к новой строке в ключевом курсоре сервер по значениям ключей выполняет запрос к используемым таблицам. Поэтому текущая строка курсора содержит реальную информацию, полученную из базы на момент перехода. Для создания ключевого курсора необходимо иметь уникальные индексы для всех используемых таблиц. При отсутствии ключевого индекса хотя бы у одной из таблиц запроса курсор преобразуется в статический. Ключевые курсоры позволяют удалять текущую строку и вносить изменения в значения ее полей. Так как набор ключей создается однократно, добавление, удаление и изменение записей в таблицах базы другими пользователями после открытия курсора могут привести к несоответствию набора ключей в курсоре и таблицах базы. Добавленные записи будут недоступны через курсор, а удаленным записям и записям с измененными ключами в курсоре по-прежнему соответствуют старые ключи. Такие ключи называются поврежденными. При попытках их использовать для доступа к данным базы генерируется ошибка;

– оптимизированный последовательный курсор (FAST_ FORWARD).

При каждом переходе по строкам снова выполняется оператор SELECT и, используя данные ранее определенной текущей строки, находит следующую строку. Таким образом, в текущей строке курсора представлено фактическое состояние данных на момент доступа. Этот курсор не позволяет вносить изменения в БД. Используется по умолчанию при создании курсоров типа READ ONLY;

– динамический курсор (DYNAMIC). Представляет собой развитие ключевого курсора. При каждом обращении к новой строке сервер перестраивает набор ключей курсора, что обеспечивает доступ к фактическим данным базы. Динамический курсор позволяет изменять данные и удалять текущую строку. Этот тип курсора используется по умолчанию при создании курсоров, не являющихся READ ONLY.

По времени существования данных курсоры разделяются:

– на локальные (LOCAL). Локальные курсоры доступны в создавшем его пакете или программном модуле и существуют до момента окончания выполнения программы;
– глобальные (GLOBAL). Глобальные курсоры сохраняются после выполнения пакета или процедуры и доступны до момента разрыва соединения, в котором они были созданы. Значение по умолчанию для этого параметра задается опцией БД CURSOR_DEFAULT {LOCAL | GLOBAL}.

Создание и обработка данных в курсоре

1. Объявление курсора. В операторе объявления курсора (DECLARE) задается его имя, тип, период существования и команда SELECT, которая определяет множество строк в курсоре.

2. Открытие курсора (OPEN). В момент открытия выполняется команда SELECT и в зависимости от типа курсора формируется набор данных или ключей для доступа к строкам.

3. Обработка строк. Переход к новой текущей строке курсора и выбор ее полей (FETCH). Этот оператор предоставляет прямой или последовательный доступ к строкам. Для последовательных курсоров обычно создается цикл обработки набора строк.

4. Закрытие курсора (CLOSE). Удаляет из базы TEMPDB сведения, созданные при открытии курсора. Доступ к данным курсора становится невозможен. Однако определение курсора сохранено и его можно повторно открыть.

5. Удаление курсора (DEALLOCATE). Удаляется определение ранее закрытого курсора, при этом освобождается занимаемая его определением память. Повторное открытие удаленного курсора возможно только после его нового объявления.

В программе одновременно может быть открыто и обрабатываться несколько разных курсоров. Также допускается вложение курсоров.

Объявление курсора в программе.

DECLARE <*имя курсора*> CURSOR [LOCAL | GLOBAL].
[FORWARD ONLY | SCROLL]
[STATIC | KEYSET | FAST_FORWARD | DYNAMIC]
[READ ONLY | SCROLL_LOCKS | OPTIMISTIC]
[TYPE_WARNING]
FOR <*оператор SELECT*> — *Запрос, определяющий строки курсора*
[FOR UPDATE [OF <*имя столбца*> [, <*имя столбца*>, …]]

При объявлении курсора указывается одна из опций READ ONLY, SCROLL_LOCKS или OPTIMISTIC, определяющая способ блокировки записей в используемых таблицах базы:

– только чтение (READ ONLY). Запрещает внесение изменений в БД через курсор. На время чтения записи устанавливается разделяемая блокировка;
– изменение строки таблицы через курсор (опции SCROLL_LOCKS или OPTIMISTIC). Значение SCROLL_LOCKS разрешает изменение данных через курсор. Для этого бло-

кируются прочитанные записи таблиц, что защищает их от изменений из других транзакций. В момент чтения строки курсора (команда FETCH) на нее устанавливается монопольная блокировка (eXclusive Lock). Если курсор открыт внутри транзакции, блокировка сохраняется до конца транзакции (COMMIT | ROLLBACK). Если курсор открыт вне транзакции, блокировка снимается при переходе на другую строку.

При значении OPTIMISTIC обновляемая строка таблицы монопольно блокируется только во время внесения изменений. Для разрешения конфликта одновременного изменения данных в момент перехода сервер сохраняет образ текущей строки (значением TimeStamp или контрольной суммы полей). Если до внесения изменений текущая строка была изменена другой транзакцией, выдается сообщение об ошибке, а текущие изменения отменяются.

Между опциями, задающими тип курсора, существует зависимость:

– статический курсор (STATIC) и оптимизированный последовательный (FAST_FORWARD) не могут быть прокручиваемыми (SCROLL);
– типы FAST_FORWARD и STATIC должны использоваться только в режиме READ_ONLY.

Если в операторе DECLARE при объявлении курсора обнаружено несоответствие параметров, то сервер самостоятельно их согласует. При указании опции TYPE_WARNING пользователю выводится сообщение «The created cursor is not of the requested type», информирующее об изменении параметров курсора.

Необязательная последняя опция FOR UPDATE объявляет курсор обновляющим, разрешая командой UPDATE изменять, а DELETE — удалять строку в таблице БД, соответствующую текущей строке курсора. Для уточнения разрешенных для изменения полей в FOR UPDATE задается список OF <*имя*

столбца> [,..]. Если параметр OF … не задан, то для изменения доступны все поля текущей строки.

Курсорные переменные

Обращение к объявленному курсору может быть непосредственно по его имени или указанием переменной курсорного типа. Для использования курсорной переменной необходимо ее сначала объявить с типом курсора: DECLARE @<*Переменная*> CURSOR, а затем присвоить значение существующего курсора: SET @<*Курсорная переменная*> = < *Имя курсора*>. Курсорную переменную можно использовать вместо имени курсора в командах: OPEN, FETCH, CLOSE и DEALLOCATE. Обычно курсорные переменные создаются для доступа к курсорам через параметры хранимых процедур.

Открытие курсора.

OPEN <*Имя курсора*> | @<*Имя курсорной переменной*>;

переход к новой строке курсора — команда FETCH;

FETCH <*Способ выбора*> FROM < *курсор* | *курсорная переменная*>

[INTO <@*переменная*>,..],

где <*Способ выбора*> одно из ключевых слов FIRST, NEXT, PRIOR, LAST, ABSOLUTE <*целая константа или переменная*>, RELATIVE <*целая константа или переменная*>, определяющих способ перехода к новой текущей строке курсора. К первой, следующей и т. д., указанием номера строки (ABSOLUTE) или заданием смещения от текущей строки (RELATIVE).

Параметр FROM <*курсор* | *курсорная переменная*> задает имя курсора или имя переменной, указывающей курсор, в котором изменяется текущая строка.

Необязательный параметр INTO содержит список имен переменных, в которые заносятся значения из полей новой текущей строки курсора.

Список переменных должен соответствовать по количеству и типам данных списку выводимых столбцов команды SELECT в объявлении курсора. Переменные могут исполь-

зоваться в программе для анализа или расчетов, зависящих от значений текущей строки.

Кроме чтения строки, оператор FETCH управляет значением глобальной переменной @@FETCH_STATUS, предназначенной для создания цикла просмотра строк.

Значение @@FETCH_STATUS определяет состояние, возникшее после перехода к новой строке:

— 0 — означает успешный переход на новую строку курсора,

— –1 — (минус 1) имела место попытка выхода за пределы курсора,

— –2 — (минус 2) попытка выбора строки с поврежденным ключом (для ключевого курсора).

Закрытие курсора CLOSE <*имя курсора*> | @<*имя курсорной переменной*>

Удаление (освобождение памяти) курсора DEALLOCATE <*имя курсора*> | @<*имя курсорной переменной*>

Изменение данных с помощью курсора

Курсоры, при декларировании которых задан параметр FOR UPDATE, могут использоваться для изменения таблицы БД. В обновляющем курсоре оператор SELECT не может содержать следующие опции: DISTINCT, TOP, ORDER BY, GROUP BY. Любое из этих значений меняет тип курсора на READ_ONLY.

Для изменения в таблице данных через курсор используется обычная команда UPDATE, в которой условие WHERE выбора обновляемых строк заменяется указанием курсора:

UPDATE <*имя таблицы*> SET <*имя изменяемого поля*> = <*значение*> [,..]

WHERE CURRENT OF [GLOBAL] <*имя курсора*> | @<*курсорная переменная*>

Новые значения присваиваются перечисленным полям той строки заданной таблицы, которая соответствует текущей строке указанного курсора. Все изменяемые поля должны принадлежать одной таблице БД.

Команда удаления строки через курсор имеет вид:

DELETE [FROM] <*имя таблицы*> WHERE CURRENT OF [GLOBAL]

<*имя курсора*> | @<*курсорная переменная*>

Из заданной таблицы удаляется строка, соответствующая текущей строке курсора.

Справочные функции курсоров

Для работы с курсором используются справочные функции и переменные.

Функция статуса курсора:

CURSOR_STATUS ('LOCAL|GLOBAL', '<*имя курсора*>') возвращает код состояния курсора:

— 0 — курсор открыт, но не содержит ни одной строки,

— 1 — курсор открыт и содержит не менее одной строки,

— –1 — курсор не открыт,

— –2 — курсор не может использоваться,

— –3 — указанный курсор не найден.

Глобальная переменная @@CURSOR_ROWS содержит число строк в рабочем наборе курсора. Значение @@CURSOR_ROWS создается в момент открытия нового курсора. Для курсора FAST_FORWARD значение @@CURSOR_ROWS = –1.

Справочную информацию по открытым в соединении курсорам возвращает системная процедура sp_cursor_list [@cursor_return=] @<*Курсорная переменная*> OUTPUT, [@cursor_scope =] 1 | 2 | 3;

Первый (выходной) параметр должен задавать переменную, в которую процедура передает курсор, каждая строка которого содержит данные одного открытого курсора. Значениями второго (входного) параметра @cursor_scope указывается тип выводимых курсоров:

— 1 — параметры выводятся только для локальных курсоров,

— 2 — параметры глобальных,

— 3 — всех открытых курсоров.

Подробная информация по справочным процедурам для действующих курсорах приведена в документации [7].

1.3.2. Динамический SQL

Во многих задачах обработки БД требуется создание программ, которые в зависимости от ситуации могут работать разными объектами базы. Стандарт SQL требует прямого указания в команде имени обрабатываемого объекта (таблицы, представления пользователя и т. д.). Для создания гибких и универсальных программ в состав Transact-SQL добавлены средства, позволяющие во время выполнения программы формировать тексты SQL-команд, а затем исполнять их на сервере [6].

Исполнить созданную в программе команду можно одним из двух способов динамического SQL:

— применением функции EXECUTE (),

— обращением к системной процедуре SP_EXECUTESQL.

Функция EXEC [UTE] (< *строковое выражение* >).

Значением строкового выражения должен быть текст, состоящий из SQL-команд. Частным случаем строкового выражения может быть строковая переменная или константа. Функция EXEC [UTE] «вычисляет» значение выражения и передает текст из SQL-команд для выполнения на сервере.

Например, пусть требуется во все пользовательские таблицы в схеме Hobby, имена которых доступны через представление INFORMATION_SCHEMA. TABLES, добавить столбец «CreateDate» типа DATETIME, предназначенный для хранения момента создания строк в этих таблицах:

— объявление переменной для имени очередной пользовательской таблицы

DECLARE @TableName VARCHAR (20);

— переменная для текста команды ALTER TABLE

DECLARE @Comm VARCHAR (100) = ''; set @Comm = 'ALTER TABLE ';

— объявление курсора для перебора пользовательских таблиц

DECLARE AddColumn CURSOR FOR SELECT Table_Name FROM INFORMATION_SCHEMA. TABLES WHERE TABLE_ SCHEMA = 'Hobby' FOR READ ONLY;

OPEN AddColumn — *открытие курсора*

— выбор имени таблицы пользователя из первой строки курсора в локальную переменную

FETCH AddColumn INTO @TableName

IF @TableName IS NOT NULL

WHILE (@@FETCH_STATUS <> –1)

BEGIN

— «вычисление» и исполнение команды ALTER TABLE

EXECUTE (@Comm + 'Hobby.'+@TableName + ' ADD CreateDate

DATETIME')

— чтение имени следующей пользовательской таблицы в @ tablename

FETCH NEXT FROM AddColumn INTO @TableName

END

CLOSE AddColumn — *закрытие курсора*

DEALLOCATE AddColumn — *удаление курсора из памяти*

При всей простоте использования функция EXEC имеет существенный недостаток — уязвимость в виде SQL-инъекций. SQL-инъекция — добавление в текст исполняемых команд дополнительного вредоносного кода. Обычный путь попадания дополнительного кода через вводимые пользователем значения переменных, используемых в тексте команды [7]. Например, если запрос содержит переменную, в которую пользователь вводит необходимое ему условие фильтрации, то появляется возможность включения в это условие исполняемого кода, как показано в следующем скрипте:

DECLARE @Comm varchar (1000) *–-переменная для текста команды SELECT*

DECLARE @CONDITION varchar (100) –- *переменная для условия фильтрации*

set @CONDITION = 'Type = "business"' — *задание условия*

SET @Comm = ' SELECT * FROM titles WHERE ' + @ CONDITION — *создание команды*

EXEC (@Comm) — *исполнение запроса без дополнительных действий*.

Однако если в переменную @CONDITION ввести значение

set @CONDITION = 'Type = "business"; DROP TABLE authors', то будет выполнен запрос и удалена таблица authors'.

Поэтому более надежным способом создания динамических SQL-команд является использование специальной процедуры:

SP_EXECUTESQL [@stmt =] <*SQL-команда с параметрами*>

[{, [@params =] N'<*описание параметра* 1> [,.. n]'}

{, [@ <*параметр1*> =] '<*значение параметра 1*' [,.. *n]*}]

При обращении к процедуре SP_EXECUTESQL в первом параметре (@stmt) в ЮНИКОДЕ записывается исполняемая SQL-команда или пакет команд, которые могут содержать параметры в виде @<*имя*>. Второй параметр (@params) должен задаваться также строкой в ЮНИКОДЕ, содержащей перечисленные через запятую все параметры, указанные в команде. Для каждого параметра указывается его имя и тип данного. Последующие элементы обращения к процедуре содержат значения для параметров команды в ключевой форме <*имя параметра*> = <*значение параметра*>.

Удобным и надежным применением SP_EXECUTESQL является использование параметров в условии запросов. Например, динамически формируемый запрос для поиска данных об авторах по значениям полей City и Contract:

DECLARE @Comm NVARCHAR (300);

DECLARE @DefParams NVARCHAR (200);

set @Comm = N'SELECT au_lname FROM authors WHERE City = @Vol_City and Contract = @Vol_Contract; ';

```
set @DefParams = N'@Vol_City VARCHAR (100), @Vol_
Contract BIT';
EXEC sp_executesql @Comm, @DefParams, @Vol_City ='Москва', @Vol_Contract = 1;
```

Команда или пакет SQL-команд, заданный в параметре процедуры SP_EXECUTESQL, компилируется в момент ее вызова. Код процедуры выполняется изолированно от вызывающей программы, поэтому объявленные в ней переменные для команд, переданных в первом параметре, недоступны. Также переменные в пакете команд SP_EXECUTESQL недоступны вызывающей программе. Кроме того, хотя это и не документировано, параметрами пакета команд не могут быть имена столбцов и таблиц. Поэтому ранее рассмотренный пример добавления столбцов с помощью функции EXECUTE () не реализуется непосредственно через параметры SP_EXECUTESQL. Для создания с помощью SP_EXECUTESQL программы с динамически изменяемыми таблицами можно построить пользовательскую процедуру, которая параметром принимает имя обрабатываемой таблицы и формирует текст команды, исполняемой в SP_EXECUTESQL без использования ее параметров. Например,

```
CREATE PROC DynTable @TableName NVARCHAR (100)
As
BEGIN
      DECLARE @Comm nvarchar (300);
      set @Comm = 'ALTER TABLE '+ @TableName + ' ADD
CreateDate1
            DATETIME';
EXECUTE SP_EXECUTESQL @Comm;
END
```

Вызов процедуры DynTable в рассмотренном ранее примере с курсором имеет вид

```
EXECUTE DynTable 'Hobby.'+@TableName;
```

1.3.3. Модули обработки данных в среде Transact-SQL

Представления базы данных

Представление (view, вид, взгляд) — объект базы, определяющий таблицу (может быть и виртуальную), заданную операторами SELECT… и их объединениями (UNION/INTERSECT/EXCEPT SELECT…) из других таблиц и представлений. При обращении к представлению запрос динамически извлекает используемые данные.

Представления реализуются:

– виртуальной таблицей, в таком случае в БД хранится запрос SELECT …; создающий таблицу;

– физически хранимой таблицей (материализованное представление), созданной заданным запросом. Для материализованного представления можно создавать индексы.

Представление — это взгляд на данные группы пользователей, решающих общую задачу и имеющих одинаковые права на данные. Представление позволяет обеспечить независимость отдельных бизнес-приложений от изменения общей логической структуры базы и является единственным способом ограничить доступ пользователя к определенным строкам базовых таблиц.

Команда SELECT… в определении представления не может использовать опции INTO и ORDER BY. При указании ORDER BY сортировка строк представления не выполняется и не участвует в определении множества строк при использовании опции TOP.

Хотя представление — хранимый в БД запрос, с представлением можно работать (с определенными ограничениями) как с обычной таблицей, например, добавлять и удалять строки, вносить изменения, которые будут приводить к изменениям данных в реальных таблицах.

Создать представление в БД можно диалоговым конструктором представлений в Management Studio, который в процес-

се диалога с пользователем создает и сохраняет запрос. Другой путь — выполнить в текущей базе команду

```
CREATE [OR ALTER] VIEW <имя представления> [(<имя столбца> [,..])]
        [WITH [ENCRYPTION] [, SCHEMABINDING]]
AS SELECT.....
    [UNION ALL | INTERSECT | EXCEPT
        SELECT.....
    .....]
    [WITH CHECK OPTION]
```

При использовании в команде SELECT… одноименных или вычисляемых столбцов задание для них новых имен обязательно.

Опция ENCRIPTION — шифрует текст оператора SELECT, чтобы защитить его от изменений и выяснения состава БД.

Опция SCHEMABINDING защищает представление, связывая его структуру со структурой используемых в нем таблиц. Эта связь контролирует изменения в структурах исходных таблиц, которые влияют на представление. С опцией SCHEMABINDING имена таблиц в операторе SELECT должны включать имена схем, а список столбцов в SELECT нельзя задать *.

Опция CHECK OPTION обеспечивает соответствие строк представления заданному в SELECT условию фильтрации. Таким образом, изменения данных в представлении, приводящие к новому множеству строк, будут отображены в представлении.

Рассмотрим пример представления данных в базе PUBS. Пусть для группы пользователей необходимо предоставить доступ к книгам авторов, проживающих в городе Oakland. Для этого создадим представление «книги_авторов_Oakland», содержащее фамилию автора, название и тип книги, для авторов, проживающих в городе Oakland. Представление должно быть защищено от разрушающих структурных изменений используемых таблиц.

```
CREATE VIEW dbo.книги_авторов_Oakland WITH
SCHEMABINDING
AS
SELECT dbo.authors.au_lname, dbo.titles.title, dbo.titles.type
    FROM dbo.authors INNER JOIN dbo.titleauthor
        ON dbo.authors.au_id = dbo.titleauthor.au_id
        INNER JOIN dbo.titles ON dbo.titleauthor.title_id = dbo.
titles.title_id
    WHERE (dbo.authors.city = 'Oakland');
```

Рассмотренная команда CREATE VIEW создает обычное представление в виде виртуальной таблицы. Представление с привязкой к схемам исходных таблиц (опция SCHEMABINDING) можно искусственно преобразовать в материализованное — хранящееся в БД. Для этого ему необходимо создать кластерный индекс, содержащий все поля индексируемого представления, который будет физически храниться в базе, а индекс использоваться при обращении к данным.

Информация о имеющихся в базе программных модулях и представлениях хранится в системном представлении SYS. SQL_MODULES. Для незашифрованных представлений столбец definition хранит текст команды CREATE VIEW.

После создания представления запрос к нему выполняется как к обычной таблице. Например, требуется вывести книги по бизнесу авторов из города Oakland: SELECT au_lname AS Фамилия, title AS Название FROM dbo.книги_авторов_Oakland WHERE type = 'business' ORDER BY Фамилия;

Представления можно использовать для внесения изменений в базу. Эти изменения выполняются в самих базовых таблицах.

Обновляемые представления должны удовлетворять следующим ограничениям:

– команда UPDATE, INSERT и DELETE должна ссылаться на столбцы только одной базовой таблицы, включенной в представление,

– изменяемые столбцы не должны быть вычисляемыми,
– команда SELECT не должна содержать параметры DISTINCT, GROUP BY, HAVING и агрегатные функции (SUM, AVG, …).

Указанные ограничения обусловлены необходимостью однозначно трассировать изменения от представления до изменяемой таблицы базы.

Пример внесения изменения в таблицу authors через ранее созданное представление. Выполняется замена фамилии автора «Gren» на «Green»

```
UPDATE dbo.книги_авторов_Oakland
    SET au_fname = 'Green' WHERE au_fname = 'Gren'
```

Обновление действует на одну строку таблицы authors.

Хотя представления создают дополнительные возможности в обработке данных, их основное применение обусловлено потребностью ограничить доступ к данным по их содержанию. Отсутствие параметров в операторе SELECT не позволяет создавать динамические представления, реализующие текущие, потребностей пользователя. Для решения подобных задач используются программные хранимые в БД модули Transact-SQL.

Хранимые модули обработки данных

Среди средств, предназначенных для разработки серверной компоненты КИС, ввиду эффективности и простоты разработки широкое распространение получили программы обработки данных, хранимые в БД и исполняемые непосредственно сервером баз данных. Современные серверы БД предлагают разнообразные средства для создания таких программ и их реализации в виде отдельных модулей различного типа. Так, в MS SQL SERVER программы обработки данных могут быть созданы на языке Transact-SQL в виде хранимых процедур, функций, триггеров [8] или функций на любом языке из группы Common Language Runtime. Наибольшие возможности создания и обработки данных предоставляют сохраняемые в БД процедуры (Stored procedure).

Хранимые процедуры на Transact-SQL

Хранимые процедуры представляют собой программные модули, записанные в БД и исполняемые на сервере по команде, поступившей от клиентского приложения или другой программы сервера. Хранимые процедуры являются удобным средством распределения вычислительной нагрузки между клиентом и сервером в системах централизованной и распределенной обработки баз данных.

Хранимая процедура создается в контекстно установленной (USE *<имя БД>*) базе данных. Поэтому при создании процедуры в ее имени не указывается ни имя сервера, ни имя базы данных.

Исходными и доступными для обработки данными для процедуры могут быть:

– любые объекты базы данных,
– входные параметры (аргументы) процедуры;

Результатом работы процедуры могут быть:

– изменения структуры базы или значений данных,
– возвращаемые в вызывающую программу:
– значения параметров, наборы строк, создаваемые операторами SELECT в коде процедуры. Информационные связи процедуры показаны на рис. 1.10.

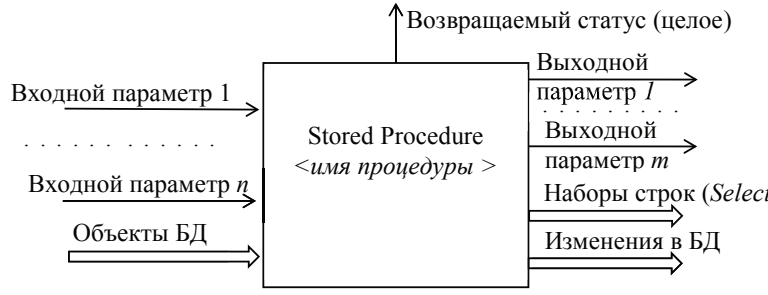


Рис. 1.10. Информационные связи хранимой процедуры

Любые программные модули являются объектами базы данных и поэтому создаются командой CREATE:

CREATE PROCEDURE [<схема>.]<*имя процедуры*> [;<*№ версии*>]

[@<*имя форм.парам.*> <*тип данных*> [=<*умалч. знач.*>] [OUTPUT]] [,.....]

[WITH [RECOMPILE] [, ENCRIPTION]

[EXEC AS {CALLER | SELF | OWNER | '<*имя польз. в БД*>'}] [,...]]]

AS

<*команды Transact-SQL*> — *код процедуры.*

Описания процедур и других модулей доступны через представление SYS. PROCEDURES, а тексты их кодов хранятся в SYS. SQL_MODULES.

Заголовок процедуры содержит:

– имя с необязательным указанием схемы, которой принадлежит процедура;
– опцию № версии — часть имени процедуры, которая задается целым положительным числом (1, 2, 3, …). Каждая версия представляет самостоятельную процедуру и может вызываться на исполнении с указанием № версии;
– необязательные формальные параметры процедуры;
– @<*имя форм. параметра*> <*тип данного*> [=<*умалчиваемое значение*>]

[OUTPUT].

Формальный параметр имеет вид локальной переменной и задается: именем (идентификатором), типом данного, опционально умалчиваемым значением, которое используется процедурой, если при вызове соответствующий фактический параметр не будет задан. Параметр, значение которого возвращается процедурой, должен иметь опцию OUTPUT. Параметры не могут иметь LOB-типы text, image.

Опция RECOMPILE требует новой компиляции при каждом вызове процедуры и используется на этапе ее отладки.

Опция ENCRIPTION требует шифрования исходного кода процедуры и таким образом защищает код и информацию о структуре данных в базе от изменений.

Опция EXEC AS {CALLER | SELF | OWNER | *'имя пользователя в БД* '} задает пользователя БД, чьи права контролируются при обращении к объектам базы:

- CALLER — процедура исполняется в контексте вызова. Вызвавший пользователь должен иметь необходимые права на процедуру и обрабатываемые в ней объекты. Этот параметр дейсвтвует по умолчанию;
- SELF — процедура исполняется в контексте (с правами) пользователя, создавшего или модифицировавшего ее модуль;
- OWNER — программа выполняется с контролем прав для текущего владельца процедуры;
- '<*имя пользователя в БД*>' — операторы в модуле выполняются в контексте указанного пользователя.

За ключевым словом «AS» располагается тело процедуры — программа, содержащая любые операторы Transact-SQL для создания/изменения объектов БД, управления данными и транзакциями, вызова хранимых процедур.

Специальный оператор RETURN [<*целое выражение*>] вычисляет значение заданного выражения и заканчивает исполнение процедуры. Значение выражения в RETURN, называемое возвращаемым статусом процедуры, обычно содержит числовой код успешности ее выполнения. Ноль, как правило, означает успешное выполнение, другие значения соответствуют коду ошибки. Возвращаемый статус передается в вызывающую программу через имя процедуры. Если в теле процедуры нет оператора RETURN, то процедура заканчивается последним выполняемым оператором.

Перед первым выполнением процедура компилируется. В результате компиляции создается план исполнения процедуры. План зависит от кода программы и набора обрабатыва-

емых объектов базы, в частности наличия и состава индексов у используемых таблиц и др. План исполнения процедуры заносится в область памяти, называемую процедурным кешем, и используется при вызове процедуры. Планы процедур, находящиеся в кеше, автоматически обновляются при каждом перезапуске SQL сервера, изменении кода процедуры командой ALTER PROCEDURE, а также перед каждым исполнением процедуры, если в заголовке задана опция RECOMPILE.

Для удаления процедур из БД используется команда DROP PROCEDURE [IF EXISTS] [<*схема*>.] <*имя процедуры*> [,..]. Удаление процедуры по имени приводит к удалению всех ее версий.

Рассмотрим пример создания хранимой процедуры.

Пусть требуется создать процедуру, возвращающую количество и все сведения об авторах (таблица authors), находящихся в том же городе, что и автор с заданным в параметре @ID идентификатором (au_id).

Запрос, который выводит сведения о таких авторах, имеет вид

```
SELECT * FROM authors WHERE city in (SELECT city FROM authors
WHERE = @ID);
```

В процедуре необходимо предусмотреть параметры:

- @ID — входной параметр — идентификатор автора (au_id),
- @count — выходной параметр (типа INT). Для вывода числа авторов, живущих в том же городе, что и заданный автор.

Создание процедуры:

```
CREATE PROCEDURE ProcAuthorsCity @ID NVARCHAR (11),
      @count INT OUTPUT
 AS
BEGIN
 SELECT * FROM authors WHERE city IN (SELECT city FROM authors
```

WHERE au_id = @ID);

SET @count=@@ROWCOUNT; — *число строк, обработанных последним SQL-оператором*

END

Вызов (исполнение) хранимых процедур.

Для исполнения процедуры из клиентского приложения или другой процедуры применяется оператор:

[EXEC [UTE]] [@<*имя переменной*> =] [[<*сервер*>.] <БД>.] <*схема*>.]

<*имя процедуры*> [; <№ версии>]

[@<*имя параметра*>=] <факт. знач. парам.>] [OUTPUT] [,...]

[WITH RECOMPILE];

Если в обращении к процедуре задано имя переменной, то ей будет присвоен статус процедуры, вычисляемый оператором RETURN. При вызове процедуры может использоваться как полное, так и сокращенное имя процедуры. Полное имя необходимо для исполнения процедур, хранящихся на другом сервере или не заданных в соединении БД. Если номер версии процедуры не задан, то по умолчанию исполняется процедура с номером 1. Фактические параметры, являющиеся входными, задаются выражениями или переменными соответствующего типа. Выходные фактические параметры задаются переменными вызывающей программы и должны сопровождаться опцией OUTPUT. Опция WITH RECOMPILE требует перестроения плана процедуры перед данным ее выполнением.

Пример вызова ранее созданной процедуры ProcAuthorsCity, возвращающей количество и все сведения об авторах, находящихся в том же городе, что и автор с заданным идентификатором. Например, задано au_id = '274–80–9391'.

DECLARE @ReturnCode INT, @к INT — *объявление переменных для кода возврата и количества найденных авторов*

— *вызов процедуры ProcAuthorsCity из ранее установленной по умолчанию базы с фактическими параметрами*

EXECUTE @ReturnCode = ProcAuthorsCity '274–80–9391', @к OUTPUT.

Результаты работы процедуры:

1. Набор строк найденных авторов:

au_id	au_lname	au_fname	…	phone
213-46-8915	Red	Marjorie	….	415 986-7020 309
274-80-9391	Straight	Dean	…	415 834-2919 5420
. . . .	. . . .	. . . .		

Возвращаемое параметром количество авторов и код возврата:

2. Select @к As [К-во], @ReturnCode As [Код возврата]

К-во	Код возврата
5	0

Функции Transact-SQL

В Transact-SQL встроено большое число готовых функций, с помощью которых преобразуются типы и обрабатываются сложные многоэлементные данные, обеспечивается аналитическая обработка результата запроса или получение системной информации. Кроме того, пользователю предоставлена возможность создать свои функции обработки данных.

Пользовательские функции

Функции образуют отдельные программные модули, но в отличие от процедуры:

– возвращают значение через свое имя и могут использоваться непосредственно в выражениях Transact-SQL,
– не могут изменять базы данных, а значит, и вызывать процедуры, изменяющие базу,
– не могут возвращать результаты через параметры,
– ошибки Transact-SQL при выполнении функции вызывают отмену ее вызова, а значит, не предусматривают обработку исключений (TRY… CATCH).

В зависимости от устойчивости результата, возвращаемого функцией, они могут быть детерминированными или неде-

терминированными. Детерминированные функции при одинаковом наборе входных значений и данных в базе в каждом вызове возвращают один и тот же результат. Недетерминированные функции могут возвращать разные результаты даже при одинаковых входных значениях, одном и том же состоянии базы. Например, используя в коде функции текущую дату, параметры сервера и др. Схема входных данных и результата для функций показана на рис. 1.11.

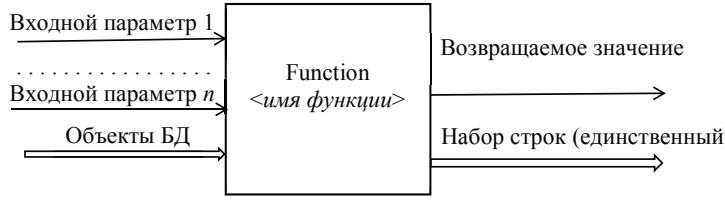


Рис. 1.11. Схема входных данных и результата функции

В зависимости от возвращаемого результата функции Transact-SQL имеют следующие типы:

- скалярные функции (SCALAR) возвращают одно значение любого типа, кроме text, ntext, image, cursor и timestamp. Скалярные функции можно вызывать в соответствующих выражениях Transact-SQL;
- однооператорные функции (INLINE). В теле таких функций должен быть записан единственный оператор SELECT. Поэтому однооператорные функции всегда возвращают значение типа TABLE и могут вызываться в предложениях FROM команд Transact-SQL;
- многооператорные функции (MULTISTATEMENT) могут содержать произвольное число операторов Transact-SQL, совместно формирующих результат типа TABLE.

Тип функции определяется типом возвращаемого значения, задаваемого параметром RETURNS в заголовке функции. Так

как функции — объекты БД, то для их создания используется команда CREATE. Как и процедура, функция создается в контексте базы.

Скалярные функции

Команда для создания или изменения скалярной функции имеет вид

CREATE [OR ALTER] FUNCTION [<*схема*>.] <*имя функции*>
([@<*имя параметра*> <*тип*> [= <*умалчиваемое значение*>] [READONLY] [,..])
RETURNS < *скалярный тип возвращаемого результата*>
[WITH [ENCRIPTION | SCHEMABINDING |
[EXEC AS {CALLER | SELF | OWNER | '<*имя пользователя*>'}] [,...]]
AS
BEGIN
<*операторы Transact SQL — тело функции*>
RETURN <*скалярное выражение заданного типа*>
[<*операторы Transact SQL* >]
END

При описании формального параметра, кроме имени, обязательно задается его тип — скалярный или табличный. Если тип параметра табличный, опция READONLY обязательна. Дополнительные параметры функции (WITH):

- ENCRIPTION обеспечит шифрование хранящего в базе исходного текста функции,
- SCHEMABINDING устанавливает запрет на изменение структуры объектов (например, таблиц), которые могут привести к нарушению работы функции.

Опция EXEC AS задает контекст безопасности, проверяемый при исполнении функции. Значение параметров такое же, как у ранее рассмотренных хранимых процедур.

Оператор RETURN <*скалярное выражение*> содержит выражение, значение которого и есть результат выполнения функции. В теле функции оператор RETURN может быть использо-

ван неоднократно. Выполнение этого оператора заканчивает выполнение функции.

Обращение (вызов) к скалярной функции записывается в выражениях Transact-SQL и имеет вид

<схема>.<имя функции> (*<фактический параметр* 1> [, …]).

Указание схемы в имени функции обязательно. Фактический параметр задается выражением, имеющим тип, соответствующий типу, заданному в формальном параметре функции.

Рассмотрим пример скалярной функции, предназначенной для контроля данных. В правильной базе публикаций PUBS сумма долей авторов для каждой книги должна составлять 100 %. Для контроля данных требуется создать функцию S_royaltyper, вычисляющую сумму долей авторов для книги, заданной первичным ключом (title_id). Долю автора в книге содержит поле royaltyper в таблице Titleauthor, как показано на рис. 1.12.

au_id	title_id	au_ord	royaltyper
267-41-2394	TC7777	2	30
472-27-2349	TC7777	3	30
672-71-3249	TC7777	1	40

Рис. 1.12. Данные о доле участия авторов в книге

Для передачи в функцию ключа книги используется входной формальный параметр @title_id. Оператор создания функции имеет вид

```
CREATE FUNCTION S_royaltyper (@title_id VARCHAR (6))
RETURNS INT
AS
BEGIN
```

DECLARE @summa INT — *локальная переменная для суммы долей авторов*

— *вычисление суммы для заданной книги*

SELECT @summa = SUM (royaltyper) FROM titleauthor

WHERE title_id = @title_id

RETURN @summa — *выход из функции*

END

Пример вызова функции S_royaltyper.

SELECT dbo.S_royaltyper ('TC7777') As [Сумма долей авторов];

Результат: сумма долей авторов

———————————.

100

Однооператорные (INLINE) функции

Однооператорная функция создает значение типа TABLE, указываемое в параметре RETURNS заголовка. Тело функции состоит из единственного оператора SELECT, создающего ее значение. Функция должна вызываться в табличных выражениях.

Например, пусть для базы PUBS необходимо получить в виде одностолбцовой таблицы ключи соавторов для автора с заданным первичным ключом (поле au_id). Поиск ключей соавторов выполняется по таблице Titleauthor. Функция имеет вид

CREATE FUNCTION Соавторы (@AU_ID VARCHAR (11))

RETURNS TABLE

RETURN SELECT TA2.au_id As [Ключи соавторов] FROM titleauthor As TA1

INNER JOIN titleauthor As TA2 ON TA1.title_id = TA2.title_id

WHERE @au_id = TA1.au_id and TA2.au_id <> @au_id;

Обращение к функции SELECT * FROM Соавторы ('724–80–9391');

Результат функции следующий:

Ключи соавторов

267-41-2394,

756-30-7391.

Многооператорные (MULTI-STATEMENT) — еще один тип табличной функции. Функция возвращает таблицу, созданную командами Transact-SQL в коде функции. Структура возвращаемой таблицы задается в заголовке параметром RETURNS путем описания табличной переменной. Значение этой переменной создается при выполнении функции:

```
CREATE [OR ALTER] FUNCTION [<схема>.] <имя функции>
([@<имя параметра> <тип данных> [=<умал. значение>]
[READONLY] [,..])
RETURNS @<имя табл. переменной>
TABLE (<столбец> <свойства> [<ограничения>..] [,.]
[<ограничения таблицы>...] [,...])
    [WITH [ENCRIPTION | SCHEMABINDING |EXEC AS ...] [,...]]
AS
    BEGIN
    <операторы Transact SQL, включая оператор RETURN>
    END
```

В теле функции заданная параметром табличная переменная наполняется данными и возвращается через имя функции при ее вызове.

Пример создания функции, которая для каждой книги вычисляет суммарное количество проданных экземпляров в заданном параметром году.

```
CREATE OR ALTER FUNCTION SaleBook (@Year INT)
RETURNS @SaleBook TABLE (Название VARCHAR (100), Количество INT)
AS
BEGIN
    INSERT @SaleBook SELECT title, SUM (qty) FROM
    titles INNER JOIN sales ON titles.title_id = sales.title_id
        WHERE YEAR (ord_date) = @Year GROUP BY title
RETURN
END
```

Запрос с обращением к функции SELECT * FROM SaleBook (2018).

Триггеры базы данных

Триггеры в базе данных — особая разновидность хранимых процедур, автоматически исполняемых:

– при выполнении команд изменения данных — DML-триггеры,
– команд изменения базы, объектов базы или сервера — DDL-триггеры.

DML- триггеры

DML-триггер принадлежит таблице или представлению и автоматически исполняется при выполнении команд INSERT, DELETE, UPDATE.

Код триггера выполняется в составе транзакции, изменяющей данные. Так как вложенные транзакции не допускают отката, в коде триггера не допускается откат внутренних транзакций. При откате транзакции, команда которой вызвала триггер, все изменения базы, выполненные триггером, также будут отменены.

Типовое назначение триггеров:

– реализация в БД бизнес-правил — выполнение дополнительных вычислений и обработки базы при изменениях в данных,
– реализации специальных ограничений на данные, обычно касающихся нескольких таблиц, или запрет определенных действий в базе, которые не могут быть заданы при помощи табличного ограничения CHECK,
– автоматическая регистрация событий, связанных с изменениями, проведенными в таблицах или структуре БД, для аудита действий пользователей, последующего анализа и исправления ошибок.

Тип триггера определяется командой, изменяющей данные:

– INSERT — триггер вставки строк в таблицу,
– DELETE — триггер удаления строк,
– UPDATE — триггер обновления (изменения) данных.

Может быть создан триггер, который срабатывает при любом наборе вышеперечисленных операций. В MS SQL SERVER

триггер для любой инициирующей команды срабатывает однократно независимо от количества обработанных строк таблицы. Для операций UPDATE и INSERT можно задать дополнительное условие, уточняющее, при изменении каких столбцов таблицы должен срабатывать триггер.

По отношению к действию, выполняемому командой, вызывающей срабатывание триггера в MS SQL SERVER, реализованы:

- триггеры AFTER (или FOR), код которых выполняется после действий инициирующей команды. Триггеры AFTER не выполняются, если действия команды вызывают нарушение табличного ограничения. Эти триггеры не могут использоваться для предотвращения ошибок, но могут выполнить дополнительные действия в БД. Одна таблица может иметь несколько триггеров AFTER;
- триггеры INSTEAD OF выполняются вместо инициирующей команды. Они могут использоваться для проверки и исправления ошибок в команде или выполнения дополнительных действий перед вставкой, обновлением или удалением строк. Другое применение таких триггеров — реализация изменений в представлениях, требующих обновлений нескольких таблиц, используемых в представлении. Если при выполнении кода триггера обнаруживаются нарушения табличных ограничений, выполняется откат действий триггера. Для каждой изменяющей команды может быть создан только один триггер INSTEAD OF.

Изменения, выполненные инициирующей командой в таблице БД, доступны в коде триггера через временные таблицы INSERTED и DELETED. Структуры таблиц INSERTED и DELETED одинаковы и совпадают со структурой таблицы, для которой создан триггер.

Таблица INSERTED содержит:

строки, добавленные инициирующей командой INSERT,

– измененные строки в их новом состоянии для команды UPDATE. INSERTED не заполняется для инициирующей команды DELETE.

Таблица DELETED хранит:

– удаленные строки при выполнении команды DELETE,
– измененные строки в их прежнем состоянии для команды UPDATE. Таблица DELETED при выполнении команды INSERT не заполняется.

Обе таблицы заполняются только при выполнении операции изменения таблицы (UPDATE), действительно изменившей данные. Таблицы INSERTED и DELETED заполняются как для триггеров AFTER, так и для INSTEAD OF. Однако в триггере INSTEAD OF вычисляемые столбцы триггерной таблицы и столбцы с типом TIMESTAMP не заполнены данными.

Любой триггер создается в контексте текущей БД командой:

CREATE [OR ALTER] TRIGGER [<*схема*>.] <*имя триггера*>

ON <*имя таблицы* | *имя представления* >

[WITH [ENCRIPTION] [EXECUTE AS <*учетная запись для проверки прав*>]]

AFTER | INSTEAD OF — *способ запуска триггера*

{[INSERT] [,] [DELETE] [,] [UPDATE]} — *инициирующие команды*

AS

<*операторы Transact SQL, составляющие код триггера*>

Для создания и модификации триггеров может использоваться редактор кода в MANAGEMENT STUDIO. Доступ к триггеру в дереве объектов БД выполняется в контекстном меню для соответствующей таблицы.

Рассмотрим пример простого триггера, не разрешающего удалять из таблицы sales сведения о продажах книг, состоявшихся в текущем году (ord_date).

В представленном ниже коде триггера, выполняемого после операции удаления (AFTER DELETE), ошибочно удаленные строки восстанавливаются в таблице sales:

```
CREATE or ALTER TRIGGER [dbo]. [NO_DEL_SALES] ON [dbo]. [sales]
    AFTER DELETE
    AS
IF EXISTS (SELECT * FROM Deleted WHERE YEAR (ord_date) >= YEAR (GETDATE ()))
    INSERT [dbo]. [sales] SELECT * FROM DELETED
        WHERE YEAR (ord_date) >= YEAR (GETDATE ());
```

Проверка триггера NO_DEL_SALES выполняется командой удаления строк из таблицы sales:

```
DELETE FROM [dbo]. [sales] WHERE ….
```

При создании программ для DML-триггеров необходимо учитывать определенные ограничения. В коде триггера разрешено использовать транзакции и любые команды Transact-SQL, кроме команд, управляющих базой данных (create database, alter database, drop database) и объектами базы (create table, create index, create rule,..).

Для триггеров обновления и вставки предусмотрено дополнительное условие срабатывания триггера, уточняющего, при изменении каких столбцов, заданных параметре SET команды UPDATE, должен выполняться код триггера. В условии используется функция UPDATE (<*имя столбца*>), которая возвращает TRUE, если указанный столбец задан в списке столбцов параметра SET команды UPDATE или INSERT, вызвавшей триггер. К функции UPDATE () можно обращаться в любом месте кода триггера.

Например, для контроля действий пользователя и защиты таблицы authors от недопустимых изменений создается триггер, контролирующий изменения ключа или фамилии автора. При попытках внесения изменений должно формироваться сообщение, содержащее учетную запись (login) пользователя, из-

менившего данные. Вид сообщения: «С ключом и именем автора работал пользователь <*login*>».

```
CREATE TRIGGER UPD_au_id ON authors
AFTER UPDATE
AS
    IF (UPDATE (au_id) OR UPDATE (au_lname))
BEGIN
DECLARE @t VARCHAR (99) = 'С ключом и именем автора работал пользователь '
    Set @t = @t + SUSER_SNAME (); — создание сообщения
    RAISERROR (@t, 16, 10) — генерация исключения
END;
```

Более гибкие условия для обнаружения в триггере обрабатываемых столбцов для команд UPDATE и INSERT предоставляет функция COLUMNS_UPDATED (). COLUMNS_UPDATED () возвращает битовую последовательность, в которой единица в двоичном разряде соответствует изменяемому столбцу. Причем в каждом байте последовательности правый (младший) бит соответствует левому столбцу.

Например, команда

```
UPDATE authors SET au_lname = 'White', contract = 1
                        WHERE au_id = '172–32–1176'
```

изменяет значение 2-го (au_lname) и 9-го (contract) столбца в таблице authors из 9 столбцов. Функция COLUMNS_UPDATED () (см. рис. 1.13) вернет шестнадцатеричный код 0x0201 [0000 0010 0000 0001], содержащий 1 во втором и девятом разрядах слева, занимая два байта. В формате целого числа результат функции для этого примера имеет значение 513.

Двоичный код, возвращаемый функцией COLUMNS_UPDATED (), может использоваться в побитовых операциях: & — «и», | — «или», ^ — исключающее «или» с двоичными кодами операндов BINARY и INT. Результат вычисления выражения имеет целое значение. Например, логическое выражение COLUMNS_UPDATED () & 513 = 513 дает TRUE для ко-

манды, изменяющей и 2-й (au_lname) и 9-й (contract) столбцы таблицы authors. А выражение COLUMNS_UPDATED () & 513 > 0 дает TRUE для команды, изменяющей либо 2-й, либо 9-й столбец. Такие выражения применяются в условных операторах для анализа в коде триггера изменяемых столбцов.

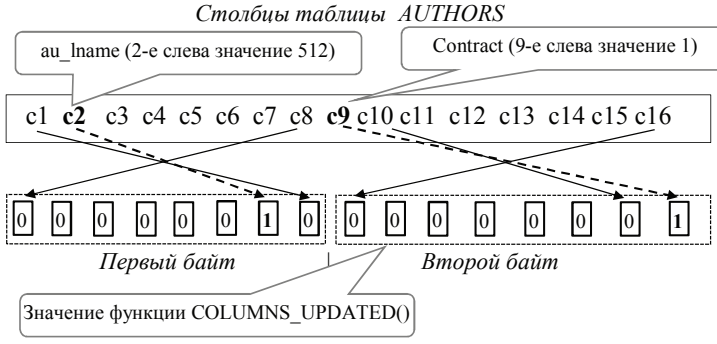


Рис. 1.13. Формирование кода функции COLUMNS_UPDATED ()

Триггеры для представлений БД

Триггеры для представления (view) часто используются для реализации операций изменения данных, которые требуют обновления нескольких таблиц, поставляющих данные в представление. Например, на основе трех таблиц authos, titleauthor и titles создано представление [ФИО автора и ID книги] содержащее:

- фамилия, имя автора, — *из authors*
- название и ключ книги, — *из titles*
- %участия и № автора в списке — *из titleauthor*

```
Create VIEW dbo. [ФИО автора и ID книги]
AS
SELECT dbo.authors.au_lname, dbo.authors.au_fname, dbo.titles.
title, dbo.titles.title_id, dbo.titleauthor.au_ord, dbo.titleauthor.
royaltyper
```

```
FROM dbo.authors INNER JOIN dbo.titleauthor
        ON dbo.authors.au_id = dbo.titleauthor.au_id INNER JOIN
dbo.titles
        ON dbo.titleauthor.title_id = dbo.titles.title_id;
```

Это представление не может быть использовано для удаления определенного автора из списка авторов конкретной книги. Так как для этого необходимо удалять запись не из представления, а удалить связывающую запись из таблицы titleauthor. Для такого удаления можно создать триггер INSTEAD OF операции DELETE, заменяющий удаление строки из представления командой удаления соответствующей записи из таблицы titleauthor:

```
Create TRIGGER [DEL_STR] ON [dbo]. [ФИО автора и ID книги]
 INSTEAD OF DELETE
AS
 DELETE FROM titleauthor
 FROM DELETED INNER JOIN Authors
 on DELETED.au_lname= authors.au_lname and
   DELETED.au_fname= authors.au_fname
    WHERE titleauthor.au_id = authors.au_id And titleauthor.
title_id =
      DELETED.title_id;
```

Проверка триггера:

```
Delete FROM dbo. [ФИО автора и ID книги] WHERE au_lname
= 'O''Leary'
and au_fname = 'Michael' and Title_id = 'BU1111';
```

DDL-триггеры

В отличие от DML DDL-триггеры срабатывают по событиям, изменяющим состав и структуру объектов БД или сервера. DDL-триггеры инициируются командами CREATE, ALTER, DROP, GRANT, DENY, REVOKE, UPDATE STATISTICS. Часть событий, связанных с этими командами, относится к базе данных, а часть — к серверу. Примеры событий, запускающих DDL-триггеры уровня сервера:

– CREATE_ DATABASE, ALTER DATABASE,……
– CREATE_LOGIN,…….

Примеры событий, запускающих DDL-триггеры на уровне БД: CREATE_TABLE, ALTER TABLE, DROP TABLE, CREATE PROCEDURE, CREATE_VIEW,….

Кроме определенных событий, относящихся к конкретному объекту, может потребоваться реакция сервера на действия со всеми объектами определенного типа. Поэтому для инициирования триггера вводятся объединяющие группы событий, касающиеся сервера в целом или базы. Например, DDL_DATABASE_LEVEL_EVENTS — все события уровня базы, DDL_TABLE_VIEW_EVENTS — любые действия с таблицами и представлениями, CREATE_FUNCTION — попытка создать новую функцию и т. д.. Полные списки событий для запуска DDL-триггера представлены в документации.

Создание или изменение DDL-триггера выполняет команда:

CREATE TRIGGER [OR ALTER] *<имя триггера>*

ON {ALL SERVER | DATABASE} — *уровень события (для БД или сервера)*

[WITH [ENCRYPTION] — *устанавливает защиту кода триггера*

EXEC AS {CALLER|SELF| 'имя Login | user'}] — *задает пользователя для проверки прав на обрабатываемые объекты*

{FOR | AFTER} — *триггер работает после операции, каскадных действий и успешной проверки ограничений*

{*< событие >* | *< группа событий >*} [,..] — *список события для запуска*

AS

<код триггера на Transact-SQL>

Опция ALL SERVER — действие триггера распространяется на сервере, независимо от установленной базы, DATABASE — действие триггера распространяется только на события в текущей БД.

Рассмотрим пример DDL-триггера, запрещающего любые изменения состава и структуры объектов текущей БД. Так

как код DDL-триггера исполняется в транзакции инициирующей команды, то для отмены действия команды в триггере достаточно выполнить откат транзакции. В примере триггер выводит сообщение и выполняет откат запрещенного действия:

```
CREATE TRIGGER DisableAction
  ON DATABASE — запуск триггера по событиям для установленной use БД
  FOR DDL_DATABASE_LEVEL_EVENTS — любая DDL-команда в для БД
AS
PRINT 'Изменения состава и структуры объектов БД запрещены!'
ROLLBACK TRANSACTION;
```

Детальную информацию о событии, вызвавшем DDL–триггер, возвращает функция EVENTDATA () в виде XML-документа следующей структуры:

```
<EventType><Тип DDL-события></EventType>
<PostTime><Время выполнения></PostTime>
<SPID><№процесса></SPID>
<ServerName><имя сервера></ServerName>
<LoginName><вход пользователя></LoginName>
<UserName><имя пользователя в БД></UserName>
<DatabaseName><имя БД></DatabaseName>
<SchemaName><имя схемы></SchemaName>
<ObjectName><имя проц./триггера></ObjectName>
<ObjectType><тип обраб.объекта БД></ObjectType>
<TSQLCommand><SQL-команда></TSQLCommand>
</EVENT_INSTANCE>
```

Функция EVENTDATA () возвращает документ только в коде DDL-триггера. В DML-триггере функция возвращает NULL. Результаты, возвращаемые функцией EVENTDATA (), могут быть выведены пользователю или сохранены в столбце таблицы типа XML.

В качестве примера рассмотрим триггер DDL_Save, записывающий в таблицу LogDDLCommand сведения об изменениях, происходящих в структуре БД.

Создание таблицы для регистрации изменений:

CREATE TABLE LogDDLCommand (F1 INT IDENTITY (1,1) PRIMARY KEY,

DDLCommand XML);

Столбец F1 — автоматический ключ (счетчик) записей, столбец DDLCommand — содержит все сведения о командах, вносивших изменения в структуру БД.

Создание триггера:

CREATE TRIGGER DDL_Save ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS — *реакция на все события БД*

AS

DECLARE @DDLCommand XML; — *объявляется переменная XML*

— для значения функции EVENTDATA ()

SET @DDLCommand = EVENTDATA ();

— запись значения EVENTDATA () в новую строку таблицы

INSERT LogDDLCommand (DDLCommand) VALUES (@DDLCommand);

Проверка триггера созданием в БД новой таблицы Tab1

CREATE TABLE Tab1 (F1 int);

Вывод из таблицы LogDDLCommand результатов регистрации изменений, записанных триггером:

```
select * From LogDDLCommand;
F1     DDLCommand
---    ------------------------------------------------
1      <EVENT_INSTANCE><EventType>
          CREATE_TABLE</EventType>
       <PostTime>2019–10–30T09:22:38....
```

В редакторе Management Studio имеется удобный (размещаемый по тегам) просмотр полей типа XML:

<EVENT_INSTANCE>

```
<EventType>CREATE_TABLE</EventType>
<PostTime>2019–10–30 T09:22:38.907</PostTime>
<SPID>52</SPID>
<ServerName>UPP </ServerName>
<LoginName>sa</LoginName>
<UserName>dbo</UserName>
<DatabaseName>Test</DatabaseName>
<SchemaName>dbo</SchemaName>
<ObjectName> DDL_Save </ObjectName>
<ObjectType>TABLE</ObjectType>
<TSQLCommand>
  <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
  ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
  ENCRYPTED="FALSE"/>
<CommandText> Create Table Tab1 (F1 int); </CommandText>
</TSQLCommand>
</EVENT_INSTANCE>
```

Несмотря на то, что триггеры предоставляют удобный инструмент реализации бизнес-правил, важно учитывать, что, выполняя неявно заданный код, триггеры дополнительно блокируют данные и выполняют фиксации и откаты транзакций. Поэтому при активном и неоправданном использовании триггеры усложняют отладку бизнес-логики и влияют на производительность сервера БД.

1.3.4. Модули обработки данных в среде CLR

Интеграция MS SQL SERVER со средой CLR

Общеязыковая среда выполнения программ является основной парадигмой платформы Microsoft.NET Framework..NET Framework предоставляет средства для создания легко взаимодействующих между собой программ различного назначения. Среда CLR предоставляет в управляемом коде различные функции и услуги, необходимые для выполнения программ, включая JIT-компиляцию, управление потоками, распределение

и освобождение памяти, обеспечение безопасности данных и программ. С другой стороны, язык Transact-SQL специально предназначен для создания и работы с БД и не является полноценным языком программирования. Поэтому в нем не поддерживаются классы, массивы, обработка двоичных кодов и другие конструкции объектно-ориентированного программирования, встроенные в управляемый код. Управляемый код, больше чем Transact-SQL подходит для реализации вычислений и выполнения сложных логических задач. NET Framework предоставляет множество готовых к использованию классов и программ доступа к файлам, шифрования и обмена данными, выполнения математических операций, работы со строками. При этом гарантируется защита и данных, и используемых программ. Кроме того, управляемый код перед выполнением компилируется в машинный, поэтому программы на языках CLR могут быть более эффективны, чем в Transact-SQL. Учитывая возможности управляемого кода для создания программ в разных сферах деятельности, среда CLR была внедрена в MS SQL SERVER [9]. Таким образом, появились средства для разработки в управляемом коде новых типов, процедур, функций и триггеров для хранения и непосредственного исполнения на сервере баз данных. Разработка CLR-модулей для MS SQL SERVER средствами Visual Studio описана в [10].

По умолчанию использование в MS SQL SERVER процедур в управляемом коде запрещено. Поэтому требуется предварительная настройка сервера, разрешающая его интеграцию с модулями управляемого кода, которая выполняется изменением конфигурации:

```
SP_CONFIGURE 'CLR ENABLED', 1;
GO;
RECONFIGURE;
```

На сервере БД модули управляемого кода хранятся в виде объектов — сборок, создаваемых из DLL-файлов. Для последующего использования (вызова) в программах обработки дан-

ных для сборки создаются хранимые процедуры, функции или триггеры Transact-SQL.

Выполняемый в SQL SERVER управляемый код предоставляет значительно большие возможности для операций с внешними объектами, а значит, несет потенциальную угрозу операционной системе и самому серверу БД. Поэтому в среде интеграции CLR с сервером предусмотрены дополнительные средства защиты самого CLR-кода и предоставляемых его операциям разрешений (CAS).

Общие правила безопасности задающие права, предоставляемые сборкам, определяются в трех разных уровнях:

– в компьютере, на котором установлен SQL SERVER, действуют права, заданные для всего управляемого кода;
– права учетной записи WINDOWS, от имени которой работает служба SQL SERVER. При этом существует возможность изменить контекст безопасности для выполняемых сборок;
– набор разрешений, заданный при создании сборки.

Выполнение операции возможно, если разрешение дано на всех перечисленных уровнях.

На уровне экземпляра сервера БД при создании сборки определяется набор предоставляемых ей прав доступа к внешним объектам. Для сборки параметром PERMISSION_SET указывается один из трех вариантов разрешений: SAVE (безопасный — без доступа к объектам вне SQL SERVER), EXTERNAL_ACCESS (разрешающий внешний доступ) и UNSAFE (небезопасный).

Сборкам с правом SAFE разрешается только локальный доступ и вычисления, выполняемые в контексте установленного соединения с сервером. Код, выполняемый сборкой не может получить доступ к внешним системным ресурсам. Права SAVE рекомендованы для сборок, выполняющих задачи вычисления и обработки данных пределах SQL SERVER.

Сборки EXTERNAL_ACCESS имеют дополнительную возможность доступа к внешним системным ресурсам: файлам, реестру, сети, переменным среды.

Сборкам UNSAFE (FullTrust) предоставляется неограниченный доступ к внутренним и системным ресурсам сервера.

Дополнительная защита от создания вредоносных сборок и от замены кода сборки обеспечивается установкой или отменой доверия сервера к БД. Для этого предназначено свойство базы данных — TRUSTWORTHY.

По умолчанию это свойство имеет значение OFF. Для создания в базе сборки ENTERNAL_ACCESS или UNSAFE она должна иметь свойство TRUSTWORTHY, а сама сборка должна быть подписана сертификатом или ассиметричным ключом. Установку параметра TRUSTWORTHY выполняет команда ALTER DATABASE *<имя БД>* SET TRUSTWORTHY ON.

CLR-функции сервера БД

CLR-функция задается статическим методом класса в сборке.NET Framework. Типами входных параметров и типом, возвращаемым скалярной функцией, может быть большинство типов SQL-сервера.

В документации на MS SQL SERVER [11] предоставлена таблица соответствия типов данных сервера и языков программирования в CLR. Для доступа к БД в управляемом коде CLR-функции используют фреймворк ADO. NET. Для обращения к данным базы используются объекты, определенные в пространстве имен Microsoft.SqlServer.Server, и объекты ADO. NET из пространства System.Data.SqlClient. Подробная информация о использовании Framework ADO. NET в управляемом коде содержится в пособии [4]. Дополнительные сведения о работе функции с БД и выполнении разных типов операций задаются в атрибуте SqlFunction [12].

Рассмотрим пример создания двух скалярных CLR-функций на языке C# в составе одной сборки.

```
//включение необходимых типов.NET Framework для работы с БД
using System;
using System.Data;
```

```
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.Data.SqlClient;//SqlClient для поставщика данных ADO. NET
```

Пример 1. Создание простой CLR-функции, выполняющей вычисления без обращения к БД. Функция CALC предназначена для перевода денежной суммы, заданной в параметре S, из долларов в рубли по курсу, заданному вторым параметром.

```
//Определяем класс CURS, в котором определена функция для перевода
public class CURS
  {[SqlFunctionAttribute ()]//определение метода как функции для SQL Server
   //Метод CALC
      public static SqlDouble CALC (SqlDouble S, SqlDouble C)
      {return S * C;}
}
```

Пример 2 представляет функцию AuthorsCount, обращающуюся к базе данных.

Функция выполняет подсчет количества строк в таблице authors.

```
//Для функции создается свой класс A_COUNT
public class A_COUNT
//Атрибут DataAccess определяет функцию только читающей данные из базы и разрешает SQL Server оптимизировать выполнение ее кода
{[SqlFunction (DataAccess = DataAccessKind.Read)]
{
public static int AuthorsCount ()
//Для работы с БД используется внутренний провайдер SQL Server
//Подготовка соединения с БД в ADO. NET
SqlConnection conn = new SqlConnection (“Data Source=…;
```

Initial Catalog=pubs; Persist Security Info=True; User ID=sa; Password= 1; “);

{conn.Open ();*//Подключение к БД*

//Создание SQL-запроса для подсчета строк в таблице authors

SqlCommand cmd = new SqlCommand (“SELECT COUNT (*) AS 'Authors Count'

FROM authors”, conn);

int N = (int)cmd.ExecuteScalar ();*//выполнение команды SELECT*

conn.Close ();*//закрытие соединения к БД*

return N;

} }

Далее эта программа транслируется в сборку и сохраняется в файле DLL (например, SQLDLL.dll). Трансляция может быть выполнена утилитой

csc/target: library/out: E:…\SQLDLL.dll E:\…\SQLDLL.dll.cs

или в MS Visual Studio.

Следующий шаг — включение сборки с состав объектов контекстно установленной базы выполняет команда:

CREATE ASSEMBLY <*имя сборки в БД*> [AUTHORIZATION <*имя ладельца*>]

FROM {*локальный или сетевой путь и имя файла сборки*}

[WITH PERMISSION_SET = {SAFE | EXTERNAL_ACCESS | UNSAFE}];

Параметр PERMISSION_SET устанавливает один из ранее рассмотренных трех типов прав, разрешенных сборке. По умолчанию для PERMISSION_SET применяется значение SAFE.

Например, создание в БД сборки из файла SQLDLL.dll

CREATE ASSEMBLY SQLDLL FROM 'E:…\SQLDLL.dll';

Чтобы функцию в управляемом коде можно было использовать в программах на Transact-SQL для каждого метода (отдельной CLR-функции), включенного в сборку, создается аналог в виде функции Transact-SQL:

CREATE [OR ALTER] FUNCTION [схема.] <*имя функции*>
({*@параметр* [AS] <*тип данных параметра*> [= <*умалчиваемое значение*>]} [,.. *n*]) RETURNS {< *скалярный тип результата*>}
[AS] EXTERNAL NAME <*имя сборки.имя класса.имя метода*>;

Особенность создания функции БД из CLR-кода в сравнении с обычной функцией Transact-SQL — указание вместо кода функции параметра EXTERNAL NAME, объявляющего функцию внешней и задающего имя выполняемого метода в сборке. Имя метода в точечной нотации содержит имя сборки, имя класса и собственное имя метода в составе класса.

Например, создание в БД функции Transact-SQL из метода CALC для пересчета денежной суммы из долларов в рубли.

```
CREATE FUNCTION dbo.My_CURS (@S FLOAT, @C FLOAT)
RETURNS FLOAT
AS
EXTERNAL NAME SQLDLL. CURS. CALC;
```

Типы для параметров и результата функции выбираются по таблице соответствия типов SQL Server и среды CLR [11].

Аналогично в БД из метода AuthorsCount создается функция для подсчета количества строк в таблице authors.

```
CREATE FUNCTION dbo.A_KOL () RETURNS INT
AS EXTERNAL NAME SQLDLL.A_COUNT. AuthorsCount;
```

Проверка функций выполняется путем их вызова из программы Transact-SQL.

```
— Вызов функции для пересчета суммы $5 в рубли по курсу 60 руб/$;
SELECT dbo.My_CURS (5,60);
— Вызов функции для подсчета числа авторов;
SELECT dbo.A_KOL ();
```

Хранимые процедуры в управляемом коде

Созданные в управляемом коде и исполняемые в MS SQL SERVER процедуры могут принимать параметры, получать

данные из базы, возвращать табличные результаты и скалярные значения через параметры.

Каждая CLR-процедура реализуется своим общим статическим методом класса. Метод может быть объявлен как viod или возвращать целое значение, интерпретируемое как код возврата из процедуры. Если метод объявлен viod, код возврата равен 0.

Параметры процедуры могут иметь любой тип из типов SQL SERVER, имеющий эквивалент в управляемом коде.

Для возврата данных из CLR-процедуры существует несколько способов:

– возврат табличных результатов — набора строк так, как это выполняет процедура на Transact-SQL,
– через выходные параметры,
– созданием сообщений.

Передачу данных подключенному клиенту из CLR-процедуры, работающей на SQL Server, выполняет объект SqlPipe, который создается при обращении к свойству Pipe объекта SqlContext (SqlContext.Pipe).

1. Метод ExecuteAndSend (<*SELECT-команда*>) объекта SqlPipe предоставляет быстрый способ передачи клиенту таблицы, полученной запросом к базе данных. Метод ExecuteAndSend отправляет результаты запроса непосредственно клиенту без использования памяти CLR-процедуры.

Далее на примере CLR-процедуры StoredProc1 показана передача клиенту результата запроса с помощью метода ExecuteAndSend.

```
public class StoredProc1
{//Процедура выполняет запрос к БД и отправляет результат клиенту
//для подключения к серверу и создания запроса использован ADO. NET
[Microsoft.SqlServer.Server.SqlProcedure]
public static void Titles ()
```

{using (SqlConnection connect = new SqlConnection ("<*строка соединения*>"))

{connect.Open ();

SqlCommand com = new SqlCommand ("SELECT title FROM titles", connect);

SqlContext.Pipe.ExecuteAndSend (com); connect.Close ();}}}

2. Другой способ передачи данных клиенту из CLR-процедуры предоставляет метод Send объекта SqlPipe. Обращение к методу Send имеет вид SqlContext.Pipe.Send. Метод Send имеет три перегрузки:

- void Send (string message) — передача клиенту сообщения в виде символьной строки,
- void Send (SqlDataRecord record) — передача одной записи,
- void Send (SqlDataReader reader) — передача набора записей (таблицы).

Использование объекта SqlDataReader в качестве источника набора строк, передаваемых клиенту методом SqlContext.Pipe.Send, показано ниже на примере класса StoredProc2.

public class StoredProc2

{*//Процедура SendReader выполняет запрос, читающий данные в объект*

//SqlDataReader и передает его значения клиенту

[Microsoft.SqlServer.Server.SqlProcedure]

public static void SendReader ()

//Здесь должно быть создание и открытие соединения с БД.....

//Создание SQL-запроса, читающего данные из базы

SqlCommand com = new SqlCommand ("SELECT au_fname, au_lname

FROM authors», <*соединение с БД*>);

SqlDataReader rd = com.ExecuteReader ();*//выполнение запроса*

SqlContext.Pipe.Send (rd);}}}*//передача результата запроса*

Как и CLR-функции CLR-процедуры, создаваемые для SQL Server, в управляемом коде транслируются и сохраняются в DLL-сборке NET Framework, а затем добавляются в базу данных командой CREATE ASSEMBLY.

Для использования в Transact-SQL для каждой CLR-процедуры командой CREATE OR ALTER PROCEDURE с параметром AS EXTERNAL NAME <*имя сборки. имя класса. имя метода*> создается своя Transact-процедура в БД.

Для вызова CLR-процедуры из Transact-SQL используется обычная команда точки входа:

EXECUTE [@<*переменная для кода возврата*>] = <*процедура*> <*параметры*>];

Контрольные вопросы

1. Каким образом в Transact-SQL выполнить частичный откат явной транзакции?
2. К какому состоянию базы приведет откат ROLLBACK TRANSACTION, выполненной в процедуре, вызываемой из явной транзакции?
3. Допустимо ли в составе обработки данных в явной транзакции создать новую базу?
4. В чем отличие ключевого курсора от статического?
5. Можно ли в составе функции Transact-SQL использовать курсор, изменяющий базу данных?
6. В чем суть SQL-инъекции?
7. Как функцию обработки БД, написанную в C#, выполнить в процедуре на Transact-SQL?

2. Хранение и обработка в БД больших двоичных объектов

Современные серверы БД поддерживают возможность хранения и транзакционной обработки данных большого объема. В MS SQL SERVER для этого предназначено несколько типов данных: VARCHAR (MAX) — текстовые данные переменной длины, использующие 8-битное кодирование, объемом до 2 Гб, NVARCHAR (MAX) — текстовые данные переменной длины в Unicode до 1 Гб символов, VARBINARY (MAX) — двоичные данные переменной длины (до 2 Гб), XML — тип для хранения документов в формате расширяемого языка разметки. Однако в корпоративной системе, помимо основной базы оперативных данных, значительная часть информации хранится в виде наборов файлов различных форматов. Большая часть этих данных создается, изменяется и используется Windows-приложениями через API-интерфейсы. При этом метаданные, описывающие содержание, права владения и доступа к таким файлам, обычно хранят в реляционной базе данных.

Для централизованного хранения и единого управления всеми корпоративными данными последние версии серверов баз данных предоставляют средства интеграции больших неструктурированных данных (BLOB) в реляционную базу. При этом сохраняется их совместимость с Windows-приложениями и предоставляется API-интерфейс для обработки файлов и каталогов.

В MS SQL SERVER реализовано несколько вариантов хранения больших двоичных объектов:

– FILESTREAM позволяет программам SQL SERVER представлять BLOB-данное полем таблицы, но хранить в отдельном файле файловой системы NTFS, обеспечивая защиту, управление и транзакционную согласованность обработки как через API-интерфейсы, так и средствами Transact-SQL;
– FILETABLE обеспечивает хранение логической структуры каталога и имен Windows-файлов и тем самым создает интегрированную среду хранения информации многих приложений и служб MS SQL SERVER, например, семантический или полнотекстовый поиск по файлу. Таблицы FileTable доступны через Transact-SQL, при этом доступ к данным из Windows-приложений сохраняется без изменений. Функциональность FILETABLE основана на хранении данных в FILESTREAM;
– удаленное хранилище больших двоичных объектов освобождает дисковую память на SQL Server, размещая большие объекты в отдельных хранилищах. Для работы приложений предоставляется API-библиотека со средствами эффективного управления удаленными данными.

2.1. Хранилище неструктурированных данных FILESTREAM

Хранилище FILESTREAM размещает большие двоичные объекты типа VARBINARY (MAX) в файловой системе NTFS. При этом управление BLOB-данными сохраняется и в Transact-SQL. Использование FILESTREAM в Transact-SQL несущественно снижает производительность сервера, так как для обработки файлов в FILESTREAM используется системный кэш ОС Windows, а не кэш-память SQL Server.

Рекомендуется использовать объекты FILESTREAM [13] вместо полей VARBINARY (MAX) если:

– средний размер объектов больше 1 МБ,
– важен быстрый доступ для чтения данных большого объема,
– целесообразно иметь как потоковый доступ к файлам из Windows-приложений, так и обработку (поиск, вставка, обновление, удаление и копирование данных) в Transact-SQL.

В иных случаях лучше хранить BLOB-объект непосредственно в поле VARBINARY (MAX) ИЛИ VARCHAR (MAX).

Управление значениями поля FILESTREAM хотя и выполняется обычными командами Transact-SQL, но имеет определенную специфику:

– так, при вставке (INSERT) записи в таблицу с полем FILESTREAM заполнение поля возможно признаком NULL, пустым значением или встроенными данными, но при этом автоматически создается файл с включенными данными;
– изменение (UPDATE) поля FILESTREAM приводит изменению данных в файле;
– удаление записи с BLOB-данными приводит к удалению связанных файлов.

2.2. Создание хранилища FILESTREAM

По умолчанию хранилище FILESTREAM недоступно. Поэтому перед началом использования FILESTREAM необходимо включить в SQL SERVER [14].

При включении разрешения требуется указать предстоящие способы обработки BLOB-данных. Возможны следующие варианты: доступ к BLOB-полям только с помощью Transact-SQL или полный доступ с помощью Transact-SQL и через API.

Для включения FILESTREAM выполняются следующие шаги:

– в меню Пуск вызовите диспетчер конфигурации SQL Server и выберите пункт «Службы SQL Server»;
– в списке служб в контексте службы SQL Server выберите команду «Открыть»;
– в соседней панели выберите экземпляр SQL Server, в котором нужно включить FILESTREAM;
– в контекстном меню для сервера выберите Свойства и перейдите на вкладку FILESTREAM, как показано на рис. 2.1;
– затем на вкладке FILESTREAM:

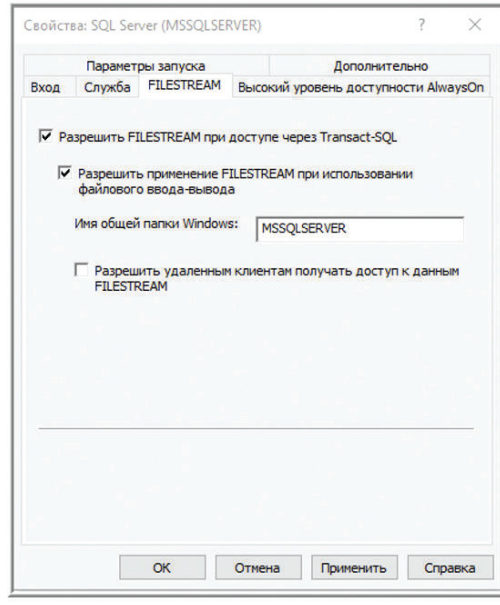


Рис. 2.1. Включение FILESTREAM на сервере

– установите флажок «Разрешить FILESTREAM при доступе через Transact-SQL»;

– если требуется работать с данными FILESTREAM из API Windows, установите флажок «Разрешить применение FILESTREAM при использовании файлового ввода-вывода»;
– введите или сохраните имя общего ресурса Windows в поле «Имя общей папки Windows»;
– нажмите кнопку «Применить».

Результат включения FILESTREAM, просмотра и изменения свойств экземпляра сервера выполняются в SQL Server Management Studio (рис. 2.2):

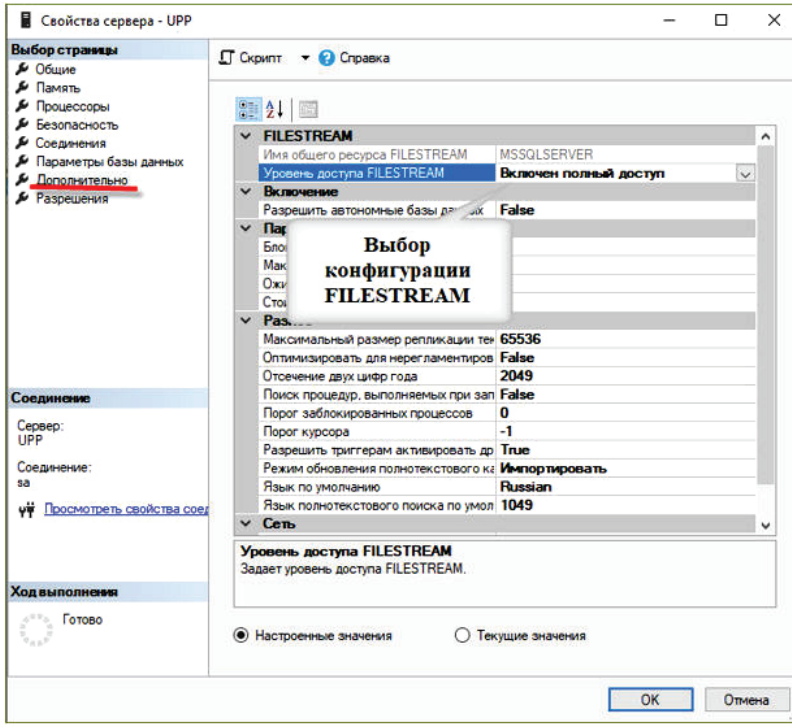


Рис. 2.2. Свойства хранилища FILESTREAM

Далее с помощью скрипта Transact-SQL требуется активировать новые параметры конфигурации сервера:

```
USE MASTER;
GO;
— включение FILESTREAM с указанием способа доступа
EXEC SP_CONFIGURE FILESTREAM_ACCESS_LEVEL, 2;
RECONFIGURE; — переконфигурация сервера.
```

Возможные значения параметра FILESTREAM_ACCESS_LEVEL в процедуре конфигурации:

- 1 — включение доступа только из Transact-SQL,
- 2 — включение доступа из Transact-SQL и в файловой системе,
- 0 — выключение.

После выполнения команды RECONFIGURE требуется перезапуск службы сервера.

После включения опции FILESTREAM на MS SQL SERVER можно создавать БД, содержащие таблицы с BLOB-данными. База, использующая хранилище FILESTREAM, требует создания отдельной файловой группы с параметром CONTAINS FILESTREAM. Файлы этой группы хранят контейнеры данных — карты соответствия BLOB-полей файлам, содержащих их значения. Эти данные используются интерфейсом между сервером БД и файловой системой.

В таблице БД хранилище больших двоичных объектов реализовано в виде столбца типа VARBINARY (MAX), а поля этого столбца будут хранится в отдельных файлах. Чтобы при создании таблицы потребовать хранения BLOB-столбца в файловой системе, для этого столбца должен быть задан дополнительный атрибут FILESTREAM, а сама таблица должна иметь уникальный идентификатор типа GUID. Размеры объектов BLOB ограничены только размером тома файловой системы, а значит, превышают ограничение 2 ГБ для данных типа VARBINARY (MAX).

Так как хранилище FILESTREAM представлено столбцом

таблицы и интегрировано в сервер БД, то для работы с BLOB-столбцами могут использоваться средства управления Transact-SQL. Для сохранности базы FILESTREAM можно использовать все команды резервного копирования и восстановления базы. Защита данных BLOB-поля осуществляется предоставлением соответствующих разрешений для таблицы или столбца таким же образом, как для любых других столбцов. Пользователь, который имеет разрешение на столбец FILESTREAM, получает доступ к связанным файлам.

Рассмотрим пример создания базы с поддержкой FILESTREAM.

```
CREATE DATABASE MyArchive
ON
PRIMARY (NAME = Arch1, FILENAME =
'E:\data\MSSQLDB\archdata.mdf'),
— файловая FileStreamGroup1 группа добавлена для хранилища FILESTREAM
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM
(name = arch2, filename = 'E:\data\mssqldb\archstream') —
для файла карты и Log
LOG ON (NAME = Archlog1, FILENAME =
'E:\data\MSSQLDB\archlog1.ldf');
GO;
```

В приведенном примере в папке E:\data\MSSQLDB будет создан каталог archstream, в котором появится файл filestream.hdr и папка журналов $FSLOG.

Если БД уже существует, то поддержка FILESTREAM (добавление файловой группы) выполняет инструкция ALTER DATABASE.

Пример модификации существующей базы для поддержки FILESTREAM (на примере базы издательства PUBS):

1. Добавление файловой группы для FILESTREAM:

```
ALTER DATABASE PUBS ADD FILEGROUP MyFileStreamGroup
CONTAINS FILESTREAM;
```

2. Добавление файла в файловую группу для поддержки FILESTREAM:

ALTER DATABASE PUBS ADD FILE (NAME = MyFileStream, FILENAME = E:\data\MSSQLDB\DIR') TO FILEGROUP MyFileStreamGroup;

После создания базы с поддержкой FILESTREAM в ней могут создаваться таблицы с BLOB-столбцами. Например, создадим таблицу архива книг с идентификационным столбцом ID и информационными столбцами для названия, количества страниц и BLOB-столбцом CONTENT для текста книги.

```
USE MyArchive
CREATE TABLE ArchiveTitles (
Id UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE
    DEFAULT NEWID (), — идентификатор записи
Название VARCHAR (40), — название книги
КоличествоСтр INT, — количество страниц
CONTENT VARBINARY (MAX) FILESTREAM NULL); — текст книги.
```

2.3. Работа с данными FILESTREAM на Transact-SQL

Так как большие двоичные данные представлены BLOB-столбцами таблиц, MS SQL SERVER поддерживает для них обычные SQL-команды. При обработке записей из параллельных подключений операции выполняются в соответствии с установленным уровнем изоляции.

1. Вставка в таблицу ArchiveTitles строки со значением NULL в поле FILESTREAM:

```
INSERT INTO ArchiveTitles (Название, КоличСтр, CONTENT)
VALUES ('Репка', 12, NULL);
```

2. Вставка следующей записи, содержащей в поле FILESTREAM строку с нулевой длиной:

```
INSERT INTO ArchiveTitles (Название, КоличСтр, CONTENT)
```

VALUES ('Русские сказки', 120, CAST (" AS VARBINARY (MAX)));

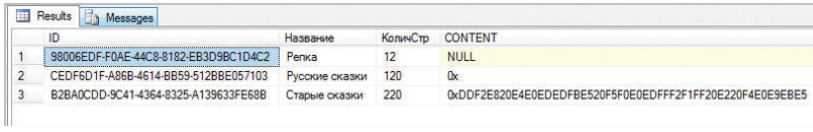
3. Вставка записи с строковыми данными в поле FILESTREAM:

FILESTREAM создает файлы Windows и помещает значение поля в файл.

INSERT INTO ArchiveTitles (Название, КоличСтр, CONTENT)
VALUES ('Старые сказки', 220,
CAST ('Эти данные хранятся в файле' AS VARBINARY (MAX)));

Вставка новых записей с определенными значениями BLOB-полей приводит к созданию файлов в выделенной папке.

4. Вывод добавленных строк (SELECT * FROM archivetitles;) имеет вид

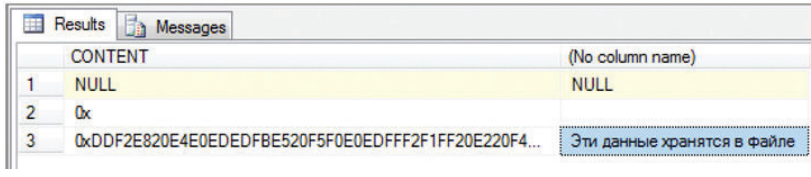


Results | Messages

	ID	Название	КоличСтр	CONTENT
1	98006EDF-F0AE-44C8-8182-EB3D9BC1D4C2	Репка	12	NULL
2	CEDF6D1F-A86B-4614-BB59-512BBE057103	Русские сказки	120	0x
3	B2BA0CDD-9C41-4364-8325-A139633FE68B	Старые сказки	220	0xDDF2E820E4E0EDEDFBE520F5F0E0EDFFF2F1FF20E220F4E0E9EBE5

5. Вывод с приведением значения поля CONTENT к исходному строковому типу:

SELECT CONTENT, CAST (CONTENT as VARCHAR (MAX))
FROM ArchiveTitles;



Results | Messages

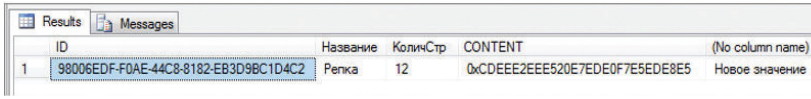
	CONTENT	(No column name)
1	NULL	NULL
2	0x	
3	0xDDF2E820E4E0EDEDFBE520F5F0E0EDFFF2F1FF20E220F4...	Эти данные хранятся в файле

6. Изменение FILESTREAM-данных (UPDATE):

UPDATE ArchiveTitles SET [Content] =
CAST ('Новое значение' as VARBINARY (MAX)) WHERE КоличСтр = 12;

Результат обновления:

SELECT *, CAST (CONTENT AS VARCHAR (MAX)) FROM ArchiveTitles
WHERE КоличСтр = 12;


Results Messages

	ID	Название	КолиЧСтр	CONTENT	(No column name)
1	98006EDF-F0AE-44C8-8182-EB3D9BC1D4C2	Репка	12	0xCDEEE2EEE520E7EDE0F7E5EDE8E5	Новое значение

Предназначенный для замены части символов в текстовом поле метод WRITE (<*Имя поля*>.WRITE) для BLOB-полей не работает. Поэтому изменять данные файлов необходимо через API.

Удаление строки, содержащей FILESTREAM-данные, выполняет обычная команда DELETE. Вместе со строкой удаляется соответствующий BLOB-объекту файл.

2.4. Загрузка в BLOB-поле произвольного файла

Особый интерес для использования BLOB-полей представляет загрузка в них содержимого файлов, имеющих различные форматы данных.

Возможность для работы MS SQL SERVER с внешними данными предоставляет табличная функция OPENROWSET (…).

Функция OPENROWSET (…) используется:

– для доступа к данным в других СУБД,
– для импорта файла в BLOB-поле.

Обращение к OPENROWSET для загрузки файла имеет вид

```
OPENROWSET (BULK 'имя файла',
                SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB)
```

При значении второго параметра SINGLE_CLOB функция OPENROWSET возвращает текстовый ASCII-файл в виде таблицы с одной строкой и одним столбцом типа VARBINARY (MAX).

Например, запрос к функции OPENROWSET:

```
SELECT * From OPENROWSET
(BULK 'E:\_upp_E\Дипломники\ВКР Елькин.docx',
      SINGLE_CLOB)
```

вернет таблицу с одним элементом:

Ниже приведен пример использования функции OPENROWSET для загрузки текстового файла в BLOB-поле:

```
DECLARE @T VARCHAR (MAX);
SELECT @T = BulkColumn
FROM OPENROWSET (BULK 'E:\_upp_E\Дипломники\ВКР Елькин.docx',
                                SINGLE_CLOB) as Text;
INSERT ArchiveTitles (Название, КоличСтр, CONTENT)
      VALUES ('Диплом Елькина', 77, CAST (@T as VARCHAR (MAX)));
```

Если при обращении к функции OPENROWSET вторым параметром задан SINGLE_BLOB, то функция возвращает поток бит, который также может быть загружен в BLOB-поле.

2.5. Поиск и обработка BLOB-полей

Если в BLOB-столбце хранятся текстовые данные, то возможна фильтрация строк таблицы по условию, содержащему регулярные выражения для анализа этих текстов. Например, поиск строк, содержащих в полях BLOB-столбца текст «Система МС» выполнит обычный запрос:

```
SELECT *, CAST (CONTENT as VARCHAR (MAX)) as ТЕКСТ
  FROM ArchiveTitles WHERE CONTENT Like %Система МС%';
```

Кроме операций Transact-SQL по включению и замене значений, доступ к BLOB-полям возможен через API-потоков в Win32. Набор функций для поля FILESTREAM предоставляет возможность получить UNC-путь к файлу, содержащему BLOB-поле. Затем, используя API-интерфейс OpenSqlFilestream, можно получить дескриптор файла и выполнить его чтение и запись функциями ReadFile () и WriteFile (). Вся обработ-

ка файла BLOB-данных выполняется в контексте транзакций SQL SERVER. Транзакция начинается в момент открытия файла и заканчивается при его закрытии.

При совместном доступе к BLOB-полю из Transact-SQL и с помощью какого-либо постороннего приложения (например, прямое обращение к текстовому файлу, содержащему BLOB-поле из Windows-блокнота) для разрешения конфликтов используется механизм версионности. При неконфликтующей последовательной обработке BLOB-поля из Transact-SQL и посторонним приложением все изменения сохраняются в обрабатываемом файле и видны в SELECT-запросе. При одновременном обращении к BLOB-данным из Transact-SQL и внешнего приложения создается файл копии, в котором сохраняется исходная версия данных.

Контрольные вопросы

1. В чем отличие хранения данных в поле VARBINARY (MAX) от VARBINARY (MAX) FILESTEAM?
2. Какие способы доступа к FILESTEAM-данным могут быть заданы при конфигурации экземпляра MS SQL SERVER?
3. При каких условиях объекты FILESTEAM предпочтительнее полей типа VARBINARY (MAX)?
4. Для каких целей в БД SQL SERVER, использующих объекты FILESTEAM, создается дополнительная файловая группа?
5. Как выполняется замена значения поля с типом FILESTEAM на NULL или на пустую строку?
6. К чему приводит попытка изменить командой UPDATE значение поля FILESTEAM во время обработки содержащего его файла посторонним приложением?
7. Какой функцией MS SQL SERVER можно загрузить файл в BLOB-поле?

3. Защита информационных ресурсов автоматизированных систем

В MS SQL SERVER используется три уровня защиты информации [15]:

– защита сервера. Возможность установить соединение с сервером предоставляется только прошедшим авторизацию пользователям, имеющим учетную запись на сервере, называемую login — вход;
– защита базы данных. Авторизованным пользователям сервера может быть дано право доступа к определенным базам. Для этого учетная запись пользователя сервера (login) связывается с ученой записью пользователя базы (user);
– защита объектов в составе БД. Легальным пользователям базы (user) даются права на выполнение определенных действий с группами или конкретными объектами. Например, право создавать других пользователей или удалять таблицы базы. Для определенных видов прав существуют правила их иерархического наследования. Например, права, выданные на схему, распространяются на объекты в схеме.

3.1. Защита баз данных

Для создания системы защиты информации в момент установки экземпляра MS SQL SERVER в нем автоматически создается учетная запись (login) устанавливающего администратора

(доменного или локального), взятая из операционной системы, например ASOIU\Popov. В процессе установки на нем может быть создана дополнительная учетная запись с именем sa (системный администратор). Обе записи включены в роль sysadmin администраторов SQL-сервера, обладающую всеми правами, в том числе правом создания новых входов в MS SQL SERVER с паролем или сертификатом для авторизации на сервере. Другой путь создания учетных записей для входа на сервер — включение записей пользователей из домена Windows в число пользователей SQL-сервера.

Контроль права на подключение к серверу может быть выполнен в одном из двух режимов:

- средствами операционной системы (Windows Authentication Mode). В этом режиме пользователи, прошедшие авторизацию в Windows, чьи учетные записи добавлены в MS SQL SERVER автоматически на основе доверительного соединения получают возможность работы в MS SQL SERVER;
- смешанный режим (Mixed Mode), Кроме доверительного соединения, к серверу могут подключиться пользователи под другими учетными записями (login) после их авторизации самим сервером БД. Смешанный режим используется для доступа к серверу как из сети Windows, так и других сетей.

Из этих двух способов авторизация средствами операционной системы является более надежной, так как может использовать методы аппаратной защиты при входе в домен, и более удобной, так как не требует повторной авторизации при подключении MS SQL SERVER.

Выбор режима авторизации удобно выполнять в Management Studio в форме свойств сервера на вкладке «Security» (рис. 3.1).

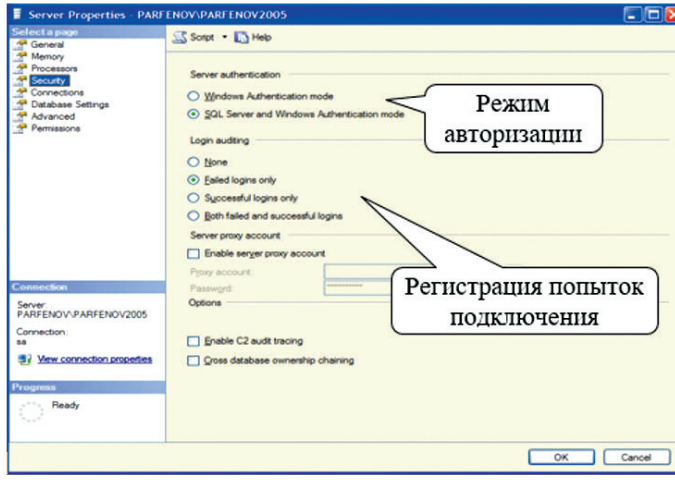


Рис. 3.1. Форма для выбора режима авторизации и аудита

В окне управления режимами в блоке Login auditing (рис. 3.1) устанавливаются правила регистрации попыток соединения с SQL-сервером. Предлагается выбрать, какие события при соединении с SQL-сервером должны регистрироваться в журнале:

– None — отсутствие регистрации попыток соединения;
– Failed — запись всех безуспешных соединений;
– Successfull — регистрация всех успешных соединений;
– Both — сохранение сведений о всех (удачных и неудачных) соединениях.

Для защиты индивидуальной информации при коллективном доступе используется дискреционный метод, основанный на разграничении доступа к данным в соответствии с назначаемыми правами (заданными разрешениями).

Для реляционных баз данных модель защиты от несанкционированного доступа основывается на трех сущностях:

– защищаемый объект (Securable) представляет любые объекты сервера и базы данных, требующие защиты.

Среди них сами серверы, базы, таблицы, схемы данных, учетные записи и роли пользователей и др.;

– субъект (Principal) — сущности, требующие и использующие ресурсы БД. К ним относятся учетные записи и роли Windows, роли сервера и БД, пользователи БД, процессы и т. д.;
– разрешения (Permissions) — права на выполнение определенных действий субъектом для объекта сервера. Например, соединение с сервером, создание, удаление, чтение данных, выполнение процедуры и т. д.

Дифференциация прав пользователей начинается с создания учетных записей для подключения к MS SQL SERVER. Новый вход на сервер может быть создан из учетной записи Windows или сертификата безопасности командой

CREATE LOGIN <*имя login в Windows*> FROM
WINDOWS | CERTIFICATE <*сертификат*>|
ASYMMETRIC KEY <*несимметричный ключ*>
[WITH [DEFAULT_DATABASE = <*имя умалчиваемой БД для подключения*>]].

Например, создание login PARFENOV\upp из учетной записи домена WINDOWS:

CREATE LOGIN [PARFENOV\upp] FROM WINDOWS.

Создание новой учетной записи для смешанного режима с авторизацией MS SQL SERVER выполняет команда с прямым указанием пароля для авторизации

CREATE LOGIN <*имя новой login* > WITH PASSWORD = '<*пароль*>'
[<*политика управления паролем*>].

Информацию о имеющихся учетных записях сервера и их свойствах возвращает системная процедура

SP_HELPLOGINS [[@LOGINNAMEPATTERN =] '< *имя login* >'].

Обращение к процедуре без указания имени записи возвращает информацию по всем входам. Удаление учетной записи сервера — команда DROP LOGIN <*имя login*>. Удаляемый вход

не должен владеть ни одним объектом сервера и быть связан с пользователем какой-либо БД.

3.2. Защита базы

Для доступа к определенной базе данных login сервера, например sa, в этой БД должен быть создан свой пользователь (user) и установлено его соответствие учетной записи сервера. Например, на сервере существуют базы с именами DB1, DB2 и входы сервера Chief и Worker. Доступ к этим базам с разными правами можно обеспечить, если создать в базах отдельных пользователей, например Vendor и Boss, и установить соответствие — дать разрешение Connect) <*login*> ↔ <*user*>. Например, как показано на рис. 3.2, вход Chief имеет доступ к обеим базам под именем и с правами пользователя Boss, а входу Worker доступна только база DB1 под с правами Vendor.

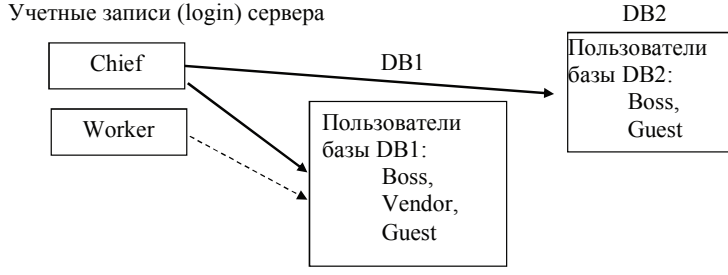


Рис. 3.2. Пример ограничения доступа пользователей к базе данных

Далее для защиты данных используется система контроля прав пользователя (user) при работе в БД. Общая схема контроля прав при доступе к БД показана на рис. 3.3.

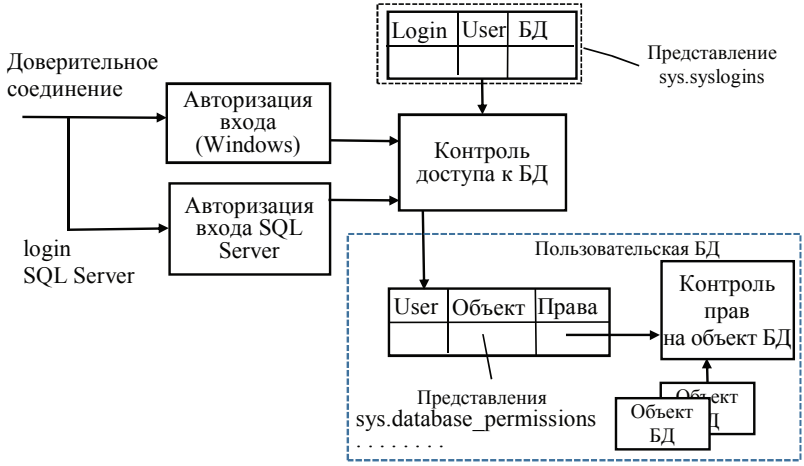


Рис. 3.3. Общая схема контроля доступа к объектам БД

При обращении к базе данных сервер проверяет наличие в ней записи user, связанной с подключившейся к серверу учетной записью login. Для связанной записи пользователя базы в каждом обращении к объекту сервер, используя системные таблицы прав, разрешает или запрещает выполнение требуемого действия.

Схемы для объектов базы данных

Для упрощения управления правами объекты БД объединяются в группы, называемые схемами. В схемы обычно включаются объекты, принадлежащие определенному пользователю. Каждый объект базы (таблица, процедура, …) принадлежит определенной схеме. Поэтому полное имя объекта в БД имеет вид

[<*имя сервера*>.] [[<*имя БД*>.] [<*имя схемы*>].]<*имя объекта*>.

При создании новой базы в ней автоматически создается схема владельца базы — dbo (DataBase Owner), а также схе-

мы для хранения системной информации: INFORMATION_SCHEMA и SYS.

Создание новой схемы пользователя базы выполняет команда Transact-SQL

CREATE SCHEMA <*имя схемы*> [AUTHORIZATION <*имя владельца*>]

[[<*команда создания таблицы | представления в новой схеме*>

| <*команда предоставления | удаления прав*>, …]];

Например, создание схемы Hobby с одновременным созданием в ней таблицы Sport выполнит следующая команда:

CREATE SCHEMA Hobby CREATE TABLE Sport (ID int, Name Char (40), …);

Для создания еще одной таблицы (Dance) в ранее созданной схеме Hobby используется команда CREATE TABLE Hobby. Dance (ID int, …).

В соединении с сервером одна из схем используется по умолчанию, если при обращении к объекту схема не задана. Указать схему по умолчанию можно SQL-командой ALTER USER <*user БД*> WITH DEFAULT_SCHEMA = <*Имя схемы*>].

Задание схемы по умолчанию не дает пользователю прав на ее объекты. Результат обращения к найденному объекту зависит от установленных прав. Если требуемого объекта нет в схеме по умолчанию, то его поиск продолжается в схеме dbo.

Управление пользователями (user) БД

Создание пользователя (user) в БД и связывание его с учетной записью сервера возможно в диалоге в Management Studio и средствами Transact-SQL. Для этого в установленной командой USE базе выполняется команда

CREATE USER <*имя польз.в БД*> FOR LOGIN <*существующий login*> |

WITHOUT LOGIN — *без связывания с login*

[WITH DEFAULT_SCHEMA = <*имя схемы по умолчанию*>]

Параметр WITHOUT LOGIN создает в базе пользователя, не связывая его с учетной записью сервера.

Удаление пользователя из БД выполняет команда DROP USER *<user БД>*.

В каждой базе автоматически создается пользователь Guest, который связан со всеми входами на сервер и предназначен для реализации минимального набора прав, необходимых любому пользователю. Во новой базе Guest исходно не имеет прав.

Права — разрешения на действия с объектами БД могут быть даны отдельному пользователю базы или группе пользователей, образующих роль. Права роли распространяются на всех ее членов. Для создания системы прав целесообразно их сгруппировать в соответствии с потребностями разных пользователей. Далее для каждого набора прав создать отдельную роль и распределить пользователей по ролям. Для администрирования права лучше назначать ролям, а не отдельным пользователям базы. Роли пользователей могут быть созданы на сервере и в каждой базе данных.

Роли сервера

На сервере автоматически создается девять предопределенных ролей. Любой (login) может быть включен в любую роль сервера. Каждая роль имеет фиксированный набор прав по обслуживанию сервера. Наиболее важными ролями являются:

- sysadmin — членам этой роли разрешены любые действия на сервере: изменение конфигурационных параметров сервера, способа авторизации, создание новых баз, создание и удаление пользователей и т. д.;
- serveradmin — права запуска и останова сервера, управление конфигурацией;
- securityadmin — создание и управление правами учетных записей сервера, просмотр журналов событий;
- public — роль, в которую включаются все учетные записи сервера.

Роли базы данных

В момент создания новой базы в ней также определены фиксированные роли, например:

- db_owner (роль владельца БД) — имеет все права в БД (создание, изменение, удаление, доступ к любому объекту базы);
- db_securityadmin (администраторы безопасности) — управление составом ролей, правами пользователей и ролей на объекты БД;
- public — роль, в которую входят все пользователи и роли БД. Роль предназначена для задания общих прав всем пользователям базы.

Фиксированные роли не охватывают всех потребностей информационной безопасности. Надежная защита данных от несанкционированного доступа строится на системе прав для ролей, создаваемых для легальных пользователей базы. В серверах БД предусмотрены разнообразные средства создания ролей, управления их составом и правами. Роли пользователей БД могут быть вложенными и включать другие роли, но не эту же роль. Создание роли и формирование ее состава может быть выполнено в диалоге Management Studio или командой Transact-SQL:

CREATE ROLE <*имя роли БД* > [AUTHORIZATION< *владелец роли*>].

Добавление или удаление новых членов в существующую роль выполняет SQL-оператор ALTER ROLE <*имя роли БД*> {ADD MEMBER <*добавляемый пользователь* | *роль БД*> | DROP MEMBER <*удаляемый пользователь* | *роль*>}.

Роли приложений

Для защиты данных, обрабатываемых программой, может потребоваться специальный набор прав. Для этого в сервере предусмотрены роли приложений. Для разных приложений могут быть созданы отдельные роли со своим набором прав.

Роль приложения можно создать в диалоге в Management Studio или командой

CREATE APPLICATION ROLE <*имя роли приложения*> WITH PASSWORD = '<*пароль*>' [, DEFAULT_SCHEMA = <*схема базы по умолчанию*>].

Роль приложения не содержит членов и не активна по умолчанию.

Для использования роли приложения программа соединяется с MS SQL SERVER под именем какой-либо учетной записи. Затем приложение активизирует свою роль, вызывая системную процедуру

SP_SETAPPROLE '<*имя активизируемой роли*>', encrypt N '<*пароль*>' |

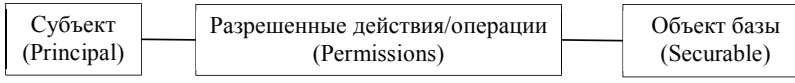
'<*пароль*>' [, NONE | ODBC].

Параметр ODBC требует использования функции защиты паролей из технологии ODBC.

Активная роль приложения отменяет все ранее установленные права, заменяя их своими разрешениями.

3.3. Разграничение доступа к данным

Дифференцированная защита информации в БД реализуется созданием бинарного отношения типа «Многое ко многим», определяющего разрешенное действие между субъектом, требующим информационного ресурса и защищаемым объектом БД.



Иерархическая модель интегрированной с Windows информационной безопасности для MS SQL SERVER представлена в табл. 4.

Таблица 4

Иерархическая модель

<table>
<tr><th>Principal
(кто, что)</th><th>Securable
(объект)</th><th>Permissions
(права)</th></tr>
<tr><td colspan="3">В ОС Windows</td></tr>
<tr><td>Учетные записи, группы Windows (доменные и локальные)</td><td>Выполняемые файлы SQL Server, скрипты и файлы БД, ключи реестра</td><td>Контроль ОС: выполнение, чтение, запись, редактирование</td></tr>
<tr><td colspan="3">В SQL Server</td></tr>
<tr><td>Учетные записи (login) Windows и SQL Server, роли SQL Server</td><td>Учетные записи SQL Server,
Сертификаты</td><td rowspan="3">Команды Transact-SQL управления правами: GRAND, REVOKE, DENY на действия (операторы SQL): Connect, Create, Alter, Drop, Control, Select, Insert, … Execute, BackUp, …</td></tr>
<tr><td colspan="2">В базе данных</td></tr>
<tr><td>Пользователи БД (user), роли БД и приложений</td><td>База данных, пользователь, роль, схема, таблица, процедура и т. д.</td></tr>
</table>

Средства управления правами пользователей и ролей

Различаются два типа прав для пользователей или ролей:

– разрешение или запрет на создание, удаление и модификацию объекта БД, которое реализуется через права на команды DDL;
– разрешение или запрет на использование объекта, назначаемые командам DML.

Управление правами выполняют команды:

GRANT — предоставление права;

REVOKE — явное лишение (отъем) ранее предоставленного права;

DENY — полный запрет на выполнение действия.

Равнозначный результат для защиты данных дает предоставление требуемого набора прав и предоставление сначала всех прав с последующей отменой лишних прав.

Предоставление прав в целом на базу данных имеет вид

USE <*имя БД*> — *указание БД*

GRANT <*permission*> [,..] — *задает разрешенные действия с БД*

TO <*db_principal*> [,..] — *перечисляет субъекты, получающие права*

[WITH GRANT OPTION] — *открывает возможность передачи этих прав*

Примеры предоставляемых прав (permission) на БД:

Control — дает все права владельца базы;

Alter — право управления структурой БД, т. е. выполнение DDL-операторов, таких как Create, Alter и т. д., требует разрешения (Connect),

SELECT, INSERT, EXEC и т. д. — по иерархии дает право выполнения соответствующего действия с объектами этой БД.

Предоставление прав на отдельные объекты в составе базы данных выполняет команда

GRANT ALL [PRIVILEGES] | — *опция ALL предоставляет все права*

| <*permission* [(*column* [,..])] [,..] > — *разрешенные SQL-команды (права)*

[ON [<*класс объекта*>::] <*securable*>] — *доступные объекты или их классы*

TO <*principal*> [,..] — *субъект, которому даются права*

[WITH GRANT OPTION] — *с правом для субъекта передачи этих прав*

Например, предоставление определенного набора прав на схему в БД имеет вид

GRANT <*permission* > [,..] ON SCHEMA:: <*имя схемы*> TO <*principal*> [,..]

Назначаемыми на схему правами могут указываться не только SQL-команды, но и комплекты предоставленных прав, например TAKE OWNERSHIP — права владельца схемы, а значит, права на все действия с объектами схемы SELECT, INSERT и т. д.

Например, предоставление пользователю базы «user_2» права на чтение всех данных в таблицах и представлениях схемы «schema_user_2» выполнит команда

GRANT SELECT ON SCHEMA:: schema_user_2 TO user_2.

Далее рассматривается полный пример создания входа на сервер, пользователя и роли в БД и управления правами этой роли в схеме БД. Скрипт выполняется в соединении, установленном системным администратором (sa).

1. Создание на сервере входа с именем loginUR1.

CREATE LOGIN loginUR1 WITH Password = '<*пароль*>',
DEFAULT_DATABASE = <*имя БД*>, CHECK_POLICY = OFF;

2. Создание в БД пользователя UR1, связанного со входом на сервер loginUR1

— выполняется в установленной по умолчанию БД

USE <*имя БД*>;

CREATE USER UR1 FOR LOGIN loginUR1;

WITH DEFAULT_SCHEMA= SCHEMA1; *—!! хотя схема еще не создана,*

— ее умолчание принимается.
Прав на эту схему пока нет!!

3. Создание в базе данных новой пользовательской роли R1:

CREATE ROLE R1;

4. Включение пользователя UR1 в роль R1:

ALTER ROLE R1 ADD MEMBER UR1;

5. Создание схемы SCHEMA1 с созданием в ней таблицы TT, владельцем которой будет sa

CREATE SCHEMA SCHEMA1
CREATE TABLE SCHEMA1.TT (F1 INT);

6. Предоставление роли R1 права на включение записей в таблицы схемы SCHEMA1

GRANT INSERT ON SCHEMA:: SCHEMA1 TO R1;

— GRANT ALL on SCHEMA:: SCHEMA1 TO R1;

— использование опции ALL вместо INSERT предоставит роли R1 все права на объекты схемы

Проверка установленных прав.

— *Подключение к серверу под именем loginUR1*

INSERT TT VALUES (1); — *добавление строки, в таблицу TT с использованием схемы, заданной по умолчанию;*

INSERT SCHEMA1.TT values (2) — *добавит следующую строку в таблицу TT.*

Попытки изменить или удалить строки из таблицы SCHEMA1.TT вызовут исключение и диагностическое сообщение сервера.

Удаление (отъем) предоставленных прав может быть выполнено в диалоге Management Studio или командой

REVOKE [GRANT OPTION FOR]

{[ALL [PRIVILEGES]] — *отъем всех прав, на указанные в ON объекты*

| *permission* [(*column* [,..])] [,..]} — *отъем отдельных перечисленных прав*

[ON [<*тип объекта*>::] *securable*] — *объект или тип*

TO *principal* [,..] — *субъекты (user или роли), у которых отнимают права*

[CASCADE] — *требует отнять права на передачу прав*

Например, отмена права создания таблиц пользователю БД с именем «User2» выполняет команда REVOKE CREATE TABLE TO User2.

Кроме предоставления и отъема прав для защиты объектов, используется команда запрета действий с объектами базы — DENY. Запрет проверяется первым, поэтому имеет приоритет над разрешениями. Запрет не допускает выполнение действий через участие пользователя в ролях, имеющих соответствующие разрешения. Для исключения полной потери объекта запрет не применяется к владельцам объектов и членам роли администратора сервера (sysadmin).

Команда запрета разрешений

DENY {[ALL [PRIVILEGES]] | — *ALL запрет на выполнение любых действий;*

permission [(*column* [,..])] [,..]} — *запрет отдельных прав;*

ON [<*класс объекта*>::] *securable*] — *защищаемый объект или класс объектов;*

TO *principal* [,..] — *субъекты, для которых устанавливается запрет;*

[CASCADE] — *устанавливает запрет на передачу разрешений.*

Команда DENY также используется для установки запретов на DDL-команды управления объектами. Например, пользователю Userlog2 должно быть запрещено создание новых схем:

DENY CREATE SCHEMA TO Userlog2.

3.4. Шифрование данных средствами Microsoft SQL Server

Разграничения прав доступа служат основой защиты данных базы. Однако, в соответствии с законодательством РФ, для конфиденциальной информации необходимо шифрование данных. Эта мера защитит данные в случае хищения базы или ее копии.

Универсальные системы для защиты компьютерной информации обеспечивают шифрование хранимых на диске файлов с расшифровыванием при передаче их использующим приложениям. Такие криптографические средства работают между приложениями и носителями информации и постоянно расходуют ресурсы сервера. Поэтому многие разработчики интегрируют средства шифрования баз данных в свои СУБД.

В Microsoft SQL Server реализовано прозрачное кодирование баз данных (Transparent Data Encryption — TDE), которое шифрует всю базу (данные и журнал). Каждая страница данных хранится на диске в зашифрованном виде и расшифровывается при чтении в оперативную память. Любое приложение, работающее с базой данных, без каких-либо изменений может использовать TDE.

Так как системы шифрования с симметричным ключом более эффективны, чем и ассиметричное шифрование в SQL Server используются обе системы шифрования. Симметричное шифрование применяется для защиты базы, а ассиметричное для защиты ключа шифрования. Кроме того, криптографическая защита баз данных Microsoft SQL Server интегрирована в систему защиты информации ОС Windows.

Для защиты данных в базах SQL Server используется система ключей [16], соответствующая стандарту ANSI X9.17. Каждая пользовательская база шифруется своим симметричным ключом — Database Encryption Key (DEK). Этот ключ защищается сертификатом или асимметричным ключом, который создается и хранится в базе MASTER.

Для защиты в БД MASTER сертификатов или асимметричных ключей используется главный ключ базы MASTER — Database Master Key (DMK).

Для защиты главного ключа базы DMK используются асимметричный ключ SMK, созданный службой шифрования Service Master Key в ОС Windows. SMK создается отдельно для каждого экземпляра MS SQL SERVER.

Иерархия использования ключей шифрования показана на рис. 3.4.

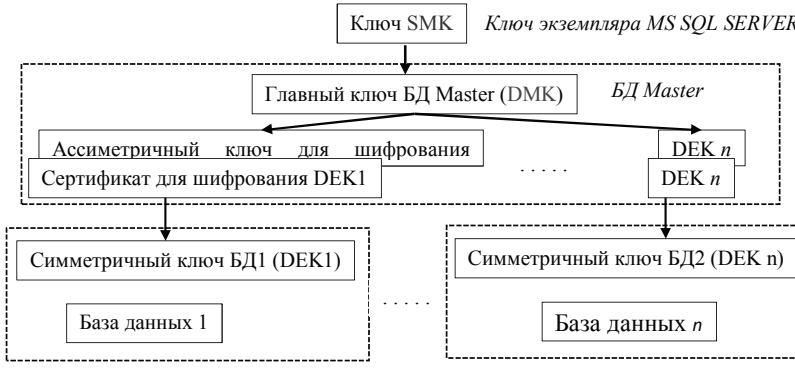


Рис. 3.4. Иерархия использования ключей шифрования

Рассмотрим пример создания ключей для шифрования базы PUBS.

```
USE Master; GO;
— Создание главного ключа БД MASTER
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '<Пароль>';
GO;
— Создание сертификата
CREATE CERTIFICATE MyCert WITH SUBJECT = 'My_DEK_Certificate';
— Здесь MyCert — имя сертификата, которое надо указать при создании
— Защищаемого симметричного ключа для БД
— Опция WITH SUBJECT требует генерации ключей для защиты ключей базы и
— Задает имя владельца для созданного и подписанного сервером сертификата
GO;
USE PUBS; GO;
— Создание симметричного ключа для БД с защитой сертификатом MyCert
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE MyCert; GO;
— AES_128 задает использование стандартного (Advanced Encryption
— Standard) алгоритма симметричного шифрования с длиной ключа 128 бит;
— При создании ключа сервер напоминает о необходимости сделать копии
— сертификата и секретного ключа для БД;
— Включение шифрования базы
ALTER DATABASE PUBS SET ENCRYPTION ON; GO;
```

При использовании шифрования базы надо иметь ввиду, что поля FILESTREAM, хранящие данные в файлах ОС, будут недоступны в Transact-SQL.

В MS SQL SERVER, кроме сплошного шифрования всей базы, возможно кодирование части данных. Наличие для БД симметричного ключа позволяет выполнять шифрование отдельных столбцов таблицы. Для этого в командах Transact-SQL при каждом обращении к шифруемому столбцу используются встроенные функции EncryptByKey () — для шифрования полей и DecryptByKey () — для расшифровывания.

Контрольные вопросы

Выберите тип бинарного отношения (1:1, 1: M, M: N) между входами (login) на сервер и пользователями (user) БД.

1. Как избежать двойной авторизации пользователя в домене Windows, которому необходимы права для работы с БД MS SQL SERVER?
2. Когда объекты БД целесообразно размещать в одной схеме?
3. Для чего введены роли для пользователей БД?
4. Как используется роль приложения?
5. В чем отличие команды, отнимающей право на определенное действие (REVOKE), от команды, устанавливающей запрет (DENY) на выполнение этого действия в БД?
6. Для чего предназначен главный ключ базы MASTER?

Список библиографических ссылок

1. Уидом Д., Гарсиа-Молина Г., Ульман Д. Системы баз данных. Полный курс. — М. : Вильямс, 2017. — 1088 с.

2. Карпова Т. С. Базы данных : учебный курс [Электронный ресурс]. — СПб. : Единая электронная образовательная среда МБИ. — URL: http://eos.ibi.spb.ru/umk/5_8/5/5_R0_T7.html (дата обращения: 20.09.2019).

3. Курсоры. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/cursors?view=sql-server-2017 (дата обращения: 20.09.2019).

4. Парфенов Ю. П. Разработка приложений для баз данных в ADO.NET : учеб. пособие. — Екатеринбург : УрФУ, 2011. — 141 с.

5. Хранимые процедуры курсора. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/system-stored-procedures/cursor-stored-procedures-transact-sql?view=sql-server-2017 (дата обращения: 20.09.2019).

6. Котелевец А. Dynamic T-SQL и как он может быть полезен [Электронный ресурс] : статья. — URL: https://habr.com/ru/post/272807/ (дата обращения: 20.09.2019).

7. Руководство по SQL-инъекциям: изучаем на примерах : пер. с англ. [Электронный ресурс] : статья // Интернет технологии. — 2018. — URL: https://www.internet-technologies.ru/articles/rukovodstvo-po-sql-inekciyam-izuchaem-na-primerah.html (дата обращения: 20.09.2019).

8. Бондарь А. Г. Microsoft SQL Server 2012 в подлиннике. — СПб. : БХВ-Петербург, 2013. — 608 с.

9. Знакомство с интеграцией CLR в SQL Server. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/sql/introduction-to-sql-server-clr-integration (дата обращения: 20.09.2019).

10. Азаров Д. Создание CRL Stored Procedure в SQL Server 2012 [Электронный ресурс] : статья. — URL: https://oxozle.com/2014/02/13/sozdanie-crl-stored-procedure-v-sql-server-2012/ (дата обращения: 20.09.2019).

11. Сопоставление данных о параметрах CLR. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/clr-integration-database-objects-types-net-framework/mapping-clr-parameter-data?view=sql-server-2017 (дата обращения: 20.09.2019).

12. Скалярные функции среды CLR. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-scalar-valued-functions?view=sql-server-2017 (дата обращения: 20.09.2019).

13. Холм Д. Так BLOB или не BLOB? [Электронный ресурс] : статья // Windows IT Pro/RE. — 2012. — № 4. — URL: https://www.osp.ru/winitpro/2012/04/13016783/ (дата обращения: 20.09.2019).

14. Включение и настройка FILESTREAM. Техническая документация Microsoft [Электронный ресурс] : статья. — URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/blob/enable-and-configure-filestream?view=sql-server-2017 (дата обращения: 20.09.2019).

15. Осетрова И. С. Администрирование MS SQL Server 2014 : учеб. пособие . — СПб. : Университет ИТМО, 2016. — 90 с.

16. Фарук Базит. Шифрование в базах данных SQL Server [Электронный ресурс] : статья // Windows IT Pro/RE. — № 5. 2013. — URL: https://www.osp.ru/winitpro/2013/05/13035359/ (дата обращения: 20.09.2019).

Оглавление

Учебное издание

Парфёнов Юрий Павлович

СРЕДСТВА УПРАВЛЕНИЯ И ЗАЩИТЫ ИНФОРМАЦИОННЫХ РЕСУРСОВ АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Редактор О. С. Смирнова
Верстка Е. В. Ровнушкиной

Подписано в печать 04.09.2020. Формат 60×84 1/16.
Бумага писчая. Цифровая печать. Усл. печ. л. 7,0.
Уч.-изд. л. 5,2. Тираж 100 экз. Заказ 202.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8 (343) 375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: 8 (343) 358-93-06, 350-58-20, 350-90-13
Факс: 8 (343) 358-93-06
http://print.urfu.ru

9 785799 630881

ПАРФЁНОВ ЮРИЙ ПАВЛОВИЧ
Доцент департамента информационных технологий и автоматики УрФУ. Руководитель проектов по дисциплинам «Базы данных», «Средства управления информационными ресурсами АС», «Постреляционные хранилища данных».
Область научных интересов: геоинформатика, компьютерное моделирование социальных процессов